Dominando JavaScript com

jQuery

PLÍNIO BALDUINO

Prefácio

-- *Ah, você programa em jQuery?*

Mesmo que essa frase irrite um desenvolvedor experiente, ela é a principal prova de que um framework definitivamente foi adotado em massa pelo mercado, ao ponto de ser confundido com uma linguagem de programação. E esse é definitivamente o caso quando falamos do jQuery - um framework que sem dúvidas revolucionou o desenvolvimento web, simplificando absurdamente suas tarefas corriqueiras.

A melhor maneira de conseguir explorar todo o potencial que o jQuery pode oferecer é conhecendo melhor JavaScript - a linguagem da web, e o ambiente onde todo esse código é executado: navegadores e páginas HTML. Vale lembrar que esse mesmo JavaScript extrapolou os limites, e se faz presente no backend, em jogos, servindo de motor para implementações de novas máquinas virtuais, emulação de hardwares antigos e até como linguagem padrão de plataformas inteiras, como Firefox OS e Windows 8!

Sim, o JavaScript passou do estigma de linguagem odiada e mal compreendida a uma das mais interessantes e essenciais. E o jQuery é certamente um dos catalisadores dessa mudança.

Neste livro, Plínio Balduino parte de um código JavaScript, que estava prestes a tornar impossível de se manter, para simplificá-lo com o uso de jQuery, enquanto apresenta o funcionamento do framework por debaixo dos panos. A aplicação JavaScript tem aquele sabor que tanto conhecemos: muita manipulação de DOM e invocações AJAX, de onde começam a brotar IFs e técnicas pouco ortodoxas (as conhecidas gambiarras) para se proteger das incompatibilidades entre navegadores e suas idiossincrasias.

Muitos fizeram apostas para o futuro das linguagens de programação. Hoje vemos que os que apostaram forte em linguagens estáticas estão observando, boquiabertos, a velocidade com que o JavaScript toma espaço no mercado. Justamente os pontos fracos da linguagem foram os que possibilitaram que ela resistisse à erosão

do tempo; sua permissividade ajudou desenvolvedores de diversas épocas a fornecerem funcionalidades que não estavam disponíveis (mais conhecidos como *shims* ou *polyfills*), e no fim das contas, fez com que um "projeto de 10 dias" com quase 20 anos de idade se tornasse nada menos do que uma das mais importantes linguagens da programação moderna.

O JavaScript veio pra ficar, e não vai desaparecer tão cedo. Como o Brendan Eich, criador da linguagem diz:

-- *"always bet on JS"*

Douglas *'qmx'* Campos

Agradecimentos

Agradeço aos meus pais, Luis e Isabel, pelo suporte de uma vida inteira e pela excelente educação.

Obrigado Paulo Silveira, Adriano Almeida e Sérgio Lopes, pela confiança ao entregarem tamanha responsabilidade nas minhas mãos e pelo apoio e senso crítico durante toda a produção deste livro.

Muito obrigado, André Noel, pela gentileza de ceder um de seus trabalhos para ilustrar este livro. Caso você não seja o André Noel e ainda não conheça o trabalho dele, recomendo fortemente uma visita ao http://www.vidadeprogramador.com.br

Thank you mr. Brendan Eich for your work and thank you for answering my questions so promptly and kindly. I'll pay you some beers next time you are in São Paulo.

Fazendo um *momento Maguila*, eu não poderia deixar de agradecer também a Alexandre Borba, André Faria, Bruno Oliveira, Diego Plentz, Douglas Campos, Eduardo Cintra, Francelino Guilherme, Igor Hercowitz, Lucas Carvalho, Marcelo Tardeli, Osni Oliveira, Ricardo Massuia, Rodrigo Lorca, Vinicius Baggio, aos amigos do GURU-SP, GURU-ES e Adapti.

Minha esposa e meus filhos sentiram o peso e preço de todo o tempo que investi nesse livro, mas ainda assim ficaram ao meu lado, apoiando-me e empurrando-me para frente. Senti muito mais a falta de vocês do que realmente aparentou.

Finalmente, agradeço também a você, leitor. Espero que você se divirta tanto lendo quanto eu me diverti escrevendo.

```
// Plínio Balduino
(enjoy)();
```

Sobre o autor

Desenvolvo softwares profissionalmente desde o século passado e já passei por quase todas as plataformas de desenvolvimento para desktop, web e mobile.

A partir de 2004 comecei a perceber o poder por trás do JavaScript e do AJAX, mas ainda estava quebrando a cabeça fazendo tudo na mão, sem a ajuda de frameworks ou bibliotecas especializadas.

De uns anos para cá, aprendendo Scheme (um dialeto LISP voltado para o aprendizado) e colaborando nos primórdios do dynjs (implementação do JavaScript para a máquina virtual do Java), consegui enxergar melhor o potencial da linguagem, e comecei a falar a respeito em eventos e conferências.

Sumário

Versão: 16.2.5

Capítulo 1

Apresentação

"Um bom começo é metade do trabalho."
– Provérbio holandês

1.1 Por que um livro sobre JavaScript e jQuery?

Existem muitos livros no mercado sobre JavaScript e jQuery, mas a maioria deles acaba pecando por não mostrar realmente a linguagem, ou por apresentar os assuntos de maneira engessada, ou simplesmente por serem manuais de referência que jogam todo o conteúdo sobre o leitor sem mostrar casos reais de uso e sem apresentar uma motivação para que seja feito daquela forma.

Obviamente, existem livros muito bons que eu recomendo fortemente, como por exemplo o *Javascript: The Good Parts* [4], de Douglas Crockford, mas que podem ser traumáticos para quem está começando agora.

Minha ideia não é abraçar o mundo e escrever o livro definitivo sobre JavaScript e jQuery, mas apresentar de maneira didática como resolver problemas cotidianos,

sem tratar o leitor como criança. Apresentar os assuntos da maneira mais fluída e agradável possível será o meu desafio.

Eu sempre gostei de livros que me apresentam uma tecnologia, mostram como utilizá-la e como sobreviver quando estou começando, e ainda continuam úteis mesmo depois de eu já passar a estar confortável naquele ambiente. É exatamente isso que eu quero que esse livro seja para você, leitor.

1.2 Como o livro está organizado

O livro começa com a história de uma loja virtual bem antiga que, por uma série de razões, não utilizou JavaScript em sua versão original. Nos capítulos 3 e 4 eu apresento a linguagem enquanto reescrevo o carrinho de compras utilizando JavaScript puro, sem bibliotecas adicionais e com toda a dificuldade de se escrever código para múltiplos browsers.

A partir do capítulo 5, apresento a biblioteca jQuery e substituo quase todo o trabalho que fizemos com JavaScript puro por chamadas mais simples.

No capítulo 6 nós vamos desenvolver uma lista de tarefas para aprendermos como manipular a fundo os elementos da sua página HTML. Aqui vamos voltar ao nosso carrinho de compras e adicionar efeitos para deixar o visual mais atraente. Voltaremos à lista de tarefas no capítulo 8, dessa vez adicionando recursos de AJAX e JSON que aprenderemos no capítulo 7, além de termos uma breve introdução ao REST e sobre como utilizar com jQuery.

Vamos falar também sobre jQuery UI, seus efeitos e componentes visuais no capítulo 9 e sobre jQuery Mobile no capítulo 10.

Finalmente, temos o que eu considero a cereja do bolo: no capítulo 11 demonstro como utilizar orientação a objetos no JavaScript, indo além do feijão com arroz e ensinando a usar prototipação, herança clássica e herança múltipla; no capítulo 12 eu apresento alguns fundamentos de programação funcional e no capítulo 13 você usa tudo o que foi mostrado no livro e aprende a criar seus próprios plugins para jQuery.

No final do livro você encontra o capítulo 14 que dá dicas de como escrever código com jQuery que execute mais rapidamente.

1.3 Algumas palavras sobre JavaScript

Das linguagens famosas e largamente usadas, talvez o JavaScript seja a menos compreendida.

Isso se deve, ao meu ver, ao preconceito que a fez ser tratada como uma versão simplificada do Java. O fato de a maioria dos criadores de páginas para Web, na época chamados de *Webmasters*, não terem um bom entendimento do que a linguagem pode fazer também auxiliou. Eles simplesmente copiavam o código de algum site e colavam na própria página.

Quando Brendan Eich desenvolveu a primeira versão do JavaScript para o browser Mozilla, em 1995, a ideia era que a linguagem realmente tivesse uma sintaxe parecida com Java, pegando emprestado até mesmo alguns objetos e métodos com nomes iguais. O próprio nome, JavaScript, foi uma jogada de marketing para que a linguagem pudesse pegar carona no sucesso do Java, quebrando a resistência das pessoas em aprender uma nova linguagem.

Por outro lado, somado a isso uma série de escolhas ruins de design, praticamente dois terços da linguagem acabaram ignorados, seja pela resistência em aprender, seja pelo simples hábito de copiar e colar, ou seja pelo fato de que o desenvolvimento para Web não era algo maduro na época.

Nas palavras de Douglas Crockford[3]:

"A maioria das pessoas que escreve código em JavaScript não é de programadora. Falta-lhes o treinamento e a disciplina para escrever bons programas. JavaScript tem é uma linguagem tão poderosa que ainda assim eles conseguem fazer algo útil. Isso deu ao JavaScript a reputação de ser usada unicamente por amadores, e que não serve para programação profissional. Isso simplesmente não é verdade."
– Douglas Crockford

Após alguns anos, a linguagem passou a ser melhor compreendida e começou a perder o status de *linguagem de brinquedo*. Isso aconteceu especialmente depois do desenvolvimento e da popularização de ferramentas como o Rhino, que permite que você execute código JavaScript na Máquina Virtual do Java; node.js, que possibilita que você desenvolva servidores com JavaScript; e finalmente o jQuery, que abstrai e simplifica muito o desenvolvimento no browser.

1.4 Lista de discussão e código fonte

Disponibilizamos uma lista de discussão sobre o livro, na qual você poderá tirar dúvidas, apontar correções, indicar melhorias e o que mais achar que for relevante para o outros leitores:

https://groups.google.com/d/forum/casadocodigo-js-jquery

O código fonte estará disponível no GitHub, no endereço abaixo:

https://github.com/pbalduino/livro-js-jquery

Sinta-se à vontade para enviar dúvidas e alterar os fontes conforme você for avançando no livro.

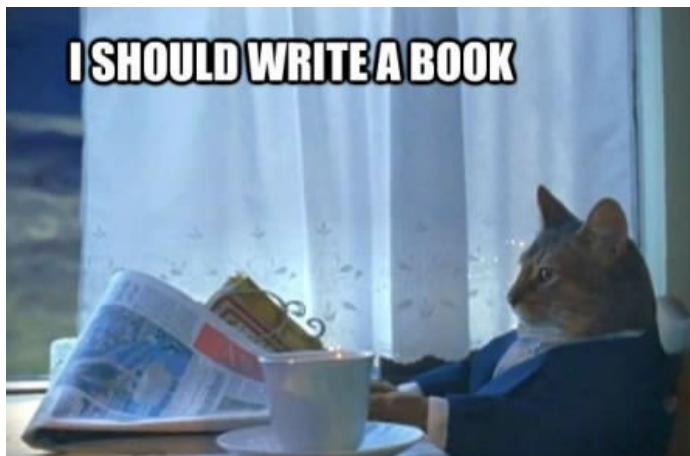



Figura 1.1: E essa história começou mais ou menos assim

Capítulo 2

Refazendo uma loja virtual

"Sou muito bom com o passado. É o presente que eu não consigo entender."
– Nick Hornby - High Fidelity

2.1 A loja virtual da Rogus Music

Nos idos de 1998, a Rogus Music, que já tinha lojas de CDs e DVDs em vários shoppings da capital, inovou e lançou uma loja virtual. A loja permitiu que seus clientes pudessem encomendar títulos importados, lançamentos e presentear sem sair de casa.

Durante anos, o dono da Rogus não se preocupou em atualizar o site, sempre dizendo que "em time que está ganhando não se mexe". O tempo passou, os filhos assumiram o negócio e a empresa passou a se chamar apenas Rogus. Hoje em dia ninguém mais compra CDs, e DVDs já não fazem mais tanto sucesso. A Rogus agora vende Blu-rays, jogos e eletrônicos.

Agora, os donos da Rogus querem atualizar o site e melhorar a experiência do usuário. Como eles têm pouca experiência com desenvolvimento para Web, resolve-

ram registrar os erros e acertos. Como disse o mais velho, “Quem sabe um dia isso não vira um livro?”.

2.2 Um site sem JavaScript

Figura 2.1: Site original da Rogus Music

A versão original do site não tem sequer uma linha de JavaScript. Segundo uma crença existente na época em que o site foi criado, JavaScript poderia infectar o computador do usuário com vírus. De qualquer maneira, os browsers ficaram muito mais seguros, e hoje em dia é improvável que algum computador seja infectado por um

código JavaScript.

Um problema que isso causa no nosso site é que cada alteração na compra do usuário, cada verificação de CPF e cada clique em um link são enviados ao servidor, sobrecarregando-o, tornando o processo mais lento e fazendo com que a experiência do usuário seja ruim, pois a todo momento a página inteira é recarregada. Você já não preencheu longos cadastros na internet nos quais, a cada próximo passo, a tela toda era renderizada, causando a impressão de lentidão?

Uma forma de resolvermos isso é deixando que o próprio browser faça uma parte do trabalho, diminuindo a carga no servidor e deixando o usuário com a impressão de que o site ficou bem mais rápido. Como um efeito colateral, o servidor vai suportar mais compras simultâneas sem a necessidade de um upgrade.

A figura 2.2 nos dá uma ideia melhor de como o servidor vai receber menos requisições.

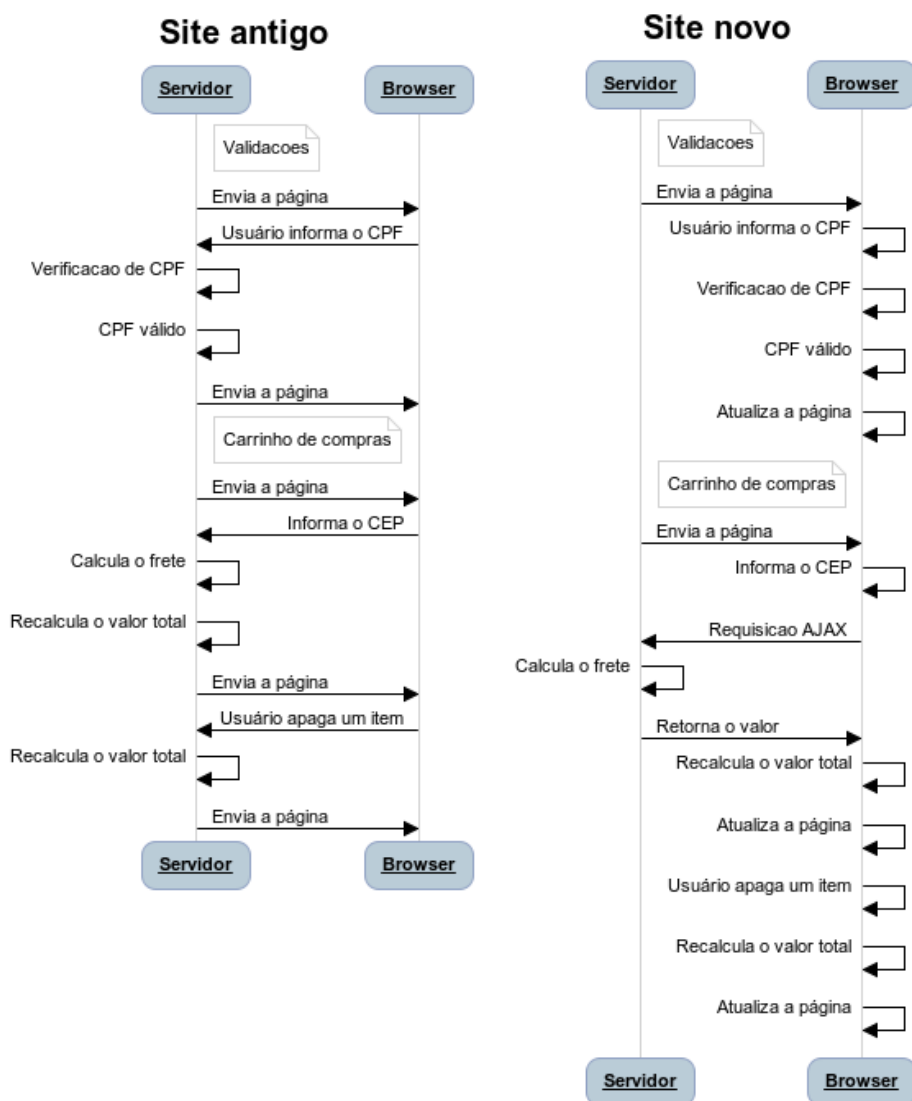



Figura 2.2: Esquema de requisições ao servidor

Nos próximos capítulos, vamos acompanhar os desafios encontrados para reescrever a loja virtual, prestando atenção no trabalho feito com o carrinho de compras.

Capítulo 3

Adicionando JavaScript

"JavaScript had to 'look like Java' only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JavaScript would have happened."
– Brendan Eich

3.1 Um novo layout

O primeiro passo a ser feito foi mudar completamente o design da tela. Podemos ver na figura 3.1 como o carrinho ficou.

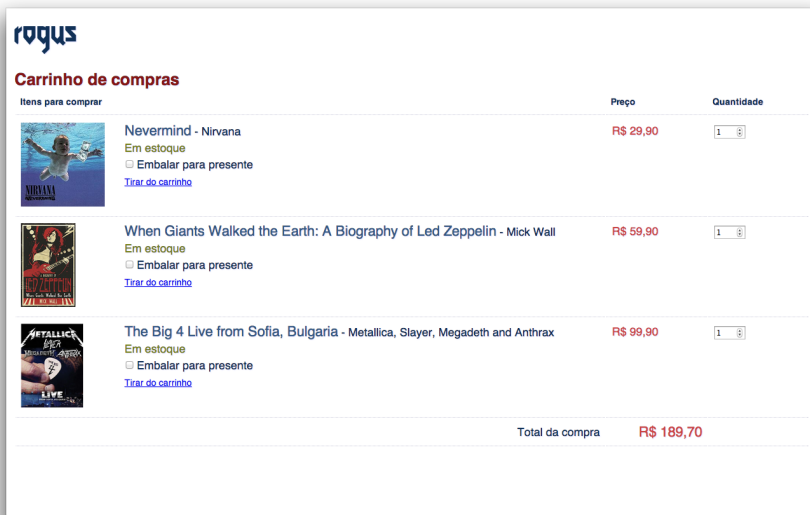



Figura 3.1: Design novo do carrinho de compras

O código do carrinho é basicamente uma tabela. Foram adicionados IDs e classes, que não existiam no layout antigo, para que possamos manipular os dados mais facilmente com JavaScript.

Listagem 3.1 - Código reduzido de um produto no carrinho:

```
<!-- -->
<table>
  <tbody>
    <tr>
      <td>
        <div>R$ 29,90</div>
      </td>
      <td>
        <input type="number">
      </td>
    </tr>
  </tbody>
  <tr>
    <td></td>
```

```
    <td>Total da compra</td>
    <td><div>R$ 29,90</div></td>
    <td></td>
  </tr>
</table>
```

Agora queremos pegar o valor de cada produto, assim como pegar o valor total da compra. Como podemos acessar cada um desses valores através do JavaScript? Precisaremos ter uma forma de identificar cada elemento do nosso documento HTML. Vamos ter que adicionar IDs e classes conforme a necessidade. Para que você entenda melhor como isso funciona, vou adicioná-los pouco a pouco, conforme formos avançando no aprendizado.

3.2 Entendendo IDs e classes

Classes e IDs são usados para identificar os elementos. Conforme veremos com mais detalhes, pense num elemento como cada pedacinho do nosso HTML formado por uma tag. É importante utilizar classes e IDs tanto para aplicarmos cores, posicionamento e formatação usando CSS quanto para localizarmos valores que queremos ler ou alterar no nosso carrinho de compras.

Um ID é um identificador único utilizado para apontar um elemento específico da nossa página. É uma boa prática nunca usar dois elementos com o mesmo ID na mesma página. Caso você use, a página vai continuar sendo desenhada e seu browser não vai explodir, mas não espere que nossos scripts funcionem corretamente. Um elemento só deve ter um ID, e um ID só deve pertencer a um elemento.

Já uma classe é utilizada para apontar um ou mais elementos. É muito usada quando você quer usar a mesma formatação para um grupo de elementos. Uma classe pode pertencer a inúmeros elementos, e um elemento pode ter inúmeras classes.

3.3 Uma questão de DOM

Na figura 3.2 nós podemos ver como o código HTML do nosso carrinho de compras 1 é representado internamente pelo browser.

Essa representação interna da página é chamada **DOM**, que significa *Document Object Model*, ou Objeto Modelo do Documento, e é criada automaticamente pelo browser toda vez que carregamos um arquivo XML ou HTML válido. Esse arquivo

é chamado de **Documento**, e cada item dentro dele (textos, imagens, botões, caixas de texto) é chamado genericamente de **Elemento**.

Quando utilizamos JavaScript para ler ou escrever dados numa página HTML, estamos na verdade manipulando o DOM. É navegando e manipulando o DOM que vamos conseguir desenvolver o nosso carrinho de compras.

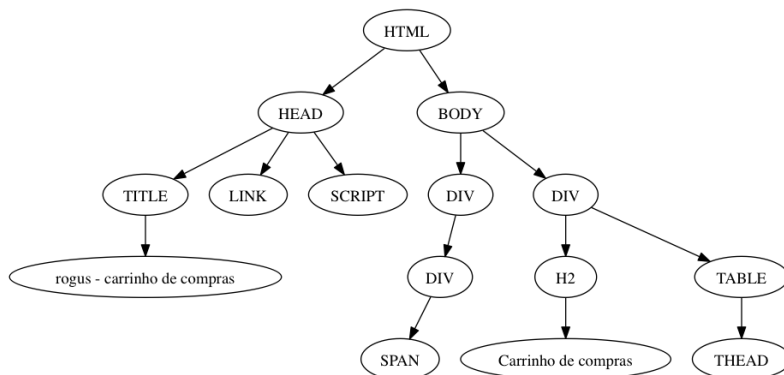
Figura 3.2: Exibindo o DOM como uma árvore

Dito tudo isso, vamos ao que realmente interessa: descobrir como calcular e alterar o valor `total` do nosso carrinho de compras.

3.4 Localizando o valor total do carrinho

Para localizar o valor total do nosso carrinho de compras, vamos adicionar um `ID` à tag `div` correspondente, ao qual vamos chamar de `total`. Relembrando, vamos usar um ID aqui porque o carrinho tem apenas um valor total, o que torna esse ID único em todo o documento.

O trecho correspondente do nosso HTML deve ficar assim:

```
<tr>
    <td></td>
    <td>Total da compra</td>
    <td><div id="total">R$ 29,90</div></td>
    <td></td>
  </tr>
</table>
```

Vamos escrever um código bem simples para exibir o valor do total das compras. Para isso, primeiro procuramos o elemento de ID `total` e, em seguida, alteramos o valor contido dentro do elemento que tem esse ID.

Localizar um elemento pelo ID é a forma mais fácil para se trabalhar com o DOM. Para isso, nós usamos a função `getElementById()` que o objeto `document` possui.

Depois de recuperado o elemento, usamos a propriedade `innerHTML`. Essa propriedade faz exatamente o que promete: retorna o texto que está entre as tags HTML que formam o elemento.

Onde colocaremos nosso JavaScript?

Após o fechamento da tag BODY, vamos abrir um espaço para escrevermos esses primeiros códigos JavaScript. Existem soluções mais elegantes e corretas, mas nesses primeiros passos vamos fazer do jeito mais rápido.

Lembre-se de **nunca** fazer isso no seu código de produção, correndo o risco de ser vítima de *bullying* por parte dos colegas.

```
<!-- -->
  </body>
  <script type="text/javascript">
     // aqui dentro vem o JavaScript
  </script>
</html>
```

```
var total = document.getElementById("total");
alert(total.innerHTML);
```

E veremos a mensagem da figura 3.3, exibindo o valor total da compra no carrinho.

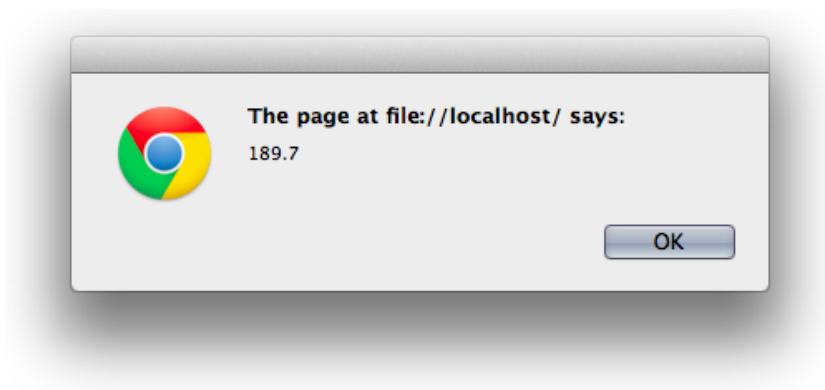



Figura 3.3: Visualizando o valor da compra

Agora vamos brincar um pouco com o valor que recebemos. **Tudo dentro de um arquivo HTML é texto**, forçando-nos muitas vezes a fazer as devidas conversões. Faremos a conversão do texto para número para poder manipular o valor em dinheiro e escrevê-lo de volta no documento.

Não há muito segredo para converter o texto em um número. Basta removermos o símbolo de moeda, substituir a vírgula por ponto e usarmos a função `parseFloat` para retornar um número. Honestamente, acho esse processo chato e repetitivo; então poderíamos aproveitar e transformar isso em uma função, que vai ficar assim:

```
function moneyTextToFloat(text) {
  var cleanText = text.replace("R$ ", "").replace(",", ".");
  return parseFloat(cleanText);
}
```

Por que usar var?

Se você já viu algum código JavaScript antes, deve ter percebido que às vezes uma variável é inicializada sem a palavra chave `var` e mesmo assim o código funciona.

Quando você inicializa a variável usando `var`, como estamos fazendo aqui, você está declarando-a no menor escopo possível, ou seja, ela é apenas uma variável local.

Quando você não utiliza `var`, a variável é criada no maior escopo possível, que geralmente é o objeto `window`, que simboliza a janela do browser que estamos usando. Isso pode permitir que outro código em outro lugar altere esse valor, tornando o comportamento do seu código imprevisível. A regra geral é: use `var` e durma tranquilo.

Podemos aproveitar e escrever uma função que faz o oposto, pegando um valor numérico e já o formatando como texto. Assim ficamos prontos para ler o valor como texto, efetuar todos os cálculos que precisarmos e gravar o valor alterado de volta no documento.

Existem inúmeras formas de se converter um valor numérico para um texto formatado. Vou usar a abordagem de multiplicar o valor por 100 e em seguida truncá-lo usando a função `Math.floor`, uma vez que vamos lidar com apenas duas casas decimais. Se você tiver necessidade de arredondar o valor para cima, use a função `Math.ceiling`. Em seguida, vou adicionar o sinal de moeda e inserir a vírgula na antepenúltima posição, usando o método `substr`. Note que, quando eu utilizo `-2` na posição do `substr`, a posição é contada do fim para o começo da String.

No final, a nossa função vai ficar com essa cara:

```
function floatToMoneyText(value) {
  var text = (value < 1 ? "0" : "") + Math.floor(value * 100);
  text = "R$ " + text;
  return text.substr(0, text.length - 2) + "," + text.substr(-2);
}
```

Se considerarmos que a função `moneyTextToFloat` faz exatamente o oposto da função `floatToMoneyText`, e vice-versa, podemos fazer um teste bem simples para avaliar se elas estão fazendo o que queremos:

```
var total = document.getElementById("total");
var formattedText = floatToMoneyText(moneyTextToFloat(total.innerHTML));

alert(formattedText === total.innerHTML);
```

Tudo isso fica melhor ainda se criarmos uma função que sirva para nos retornar o valor do total do carrinho, sem que precisemos ficar repetindo código. Usando o que já fizemos acima, a função ficaria assim:

```
function readTotal() {
  var total = document.getElementById("total");
  return moneyTextToFloat(total.innerHTML);
}
```

3.5 Usando === e ==

Note que, para efetuarmos a comparação, usamos `===` (três sinais de igual), e não `==` (dois sinais de igual). Isso acontece porque o operador `==`, ao contrário de outras linguagens baseadas na sintaxe do C, **significa equivalência, e não igualdade**.

Isso quer dizer que, quando você usa `==`, o JavaScript tenta converter um dos valores para o tipo do outro, como podemos ver abaixo. *A linha iniciada por => indica o resultado da expressão e não deve ser digitada:*

```
2 == 2;
=> true

1 == "1";
=> true

0 == [];
=> true

0 == "";
=> true
```

Porém, quando você utiliza `===`, o JavaScript não converte tipos e verifica a igualdade dos valores, sem truques:

```
2 === 2;
=> true
```

```
1 === "1";
=> false

0 === [];
=> false

0 === "";
=> false
```

A menos que você tenha um motivo realmente bom para utilizar o operador de equivalência, use sempre `===` para evitar que erros estranhos apareçam para puxar o seu pé. Da mesma forma, caso você queira verificar valores diferentes, utilize `!==` (com dois sinais de igual) ao invés de `!=` (com um sinal de igual).

3.6 Alterando o campo do total

Para alterar o valor do total do carrinho, vamos criar uma função que receba um valor numérico, formate-o usando o código que já criamos e o escreva no espaço reservado para o valor total. Faremos isso usando a já conhecida propriedade `innerHTML`, que também permite que você atribua um valor. Vamos chamar nossa função de `writeTotal`.

```
function writeTotal(value) {
  var total = document.getElementById("total");
  total.innerHTML = floatToMoneyText(value);
}
```

Apenas para testar os limites do nosso código, vamos usar o valor de `PI` como total da compra:

```
writeTotal(3.14159);
```

E finalmente podemos ver o nosso valor total formatado e alterado na figura 3.4

Total da compra R$ 3,14

Figura 3.4:

3.7 Incluindo um arquivo JavaScript na página

Agora que temos algumas funções prontas e alguma funcionalidade, é uma boa prática separarmos o código JavaScript em um outro arquivo. Vamos chamá-lo de `rogus.js`, e o gravamos no diretório `javascripts`.

Para adicionar esse arquivo na nossa página, vamos adicionar a linha abaixo dentro da seção `head` do nosso código:

```
<script src='javascripts/rogus.js' type='text/javascript'></script>
```

Podemos apagar aquela seção `<script type="text/javascript"></script>` que ficou vazia logo após o fechamento da tag `body`.

Vamos também retirar do nosso código JavaScript a linha com o `alert` e a linha em que alteramos o valor total do carrinho para `PI`. Com isso, ficamos apenas com declarações de funções dentro do código.

3.8 Brincando com o código de um jeito mais profissional

Encher o seu código de `alert` é uma péssima prática que só tinha justificativa quando se desenvolvia para *Netscape* e *Internet Explorer 5*. Atualmente, browsers modernos como o *Firefox*, o *Google Chrome* e o *Opera* têm ferramentas muito boas para depuração e verificação de código JavaScript de modo transparente.

Para o *Firefox*, você vai precisar do *Firebug*, que pode ser baixado em https://getfirebug.com/. Já o *Chrome*3.5, com os *Developer Tools*, e o *Opera*, com o *Drangonfly*, já vêm com ferramentas de desenvolvimento embutidas e não exigem a instalação de extensões adicionais.

O Internet Explorer passou a ter algumas ferramentas de desenvolvimento embutidas a partir da versão 8, mas ainda não são tão maduras e eficientes quanto as dos browsers acima.

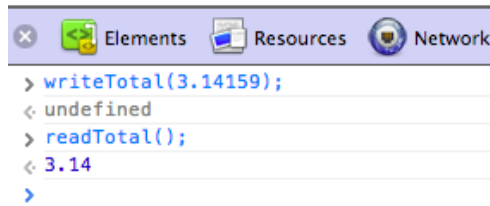



Figura 3.5: Console JavaScript no Chrome

Utilizando o browser de sua preferência, vamos usar as funções que já criamos para manipular o valor total do carrinho de compras, para em seguida exibir o mesmo valor para demonstrar que nossa alteração funcionou. **Sempre que utilizarmos o Console, considere que a linha iniciada por => é o resultado do comando, e não deve ser digitada.**

```
readTotal();
=> 189.7

writeTotal(123.45);
=> undefined

readTotal();
=> 123.45
```

O total do carrinho aparecerá alterado automaticamente na nossa página, conforme podemos ver na imagem 3.6.

Figura 3.6: Total alterado pelo nosso código

3.9 Meu browser não tem console. E agora?

Numa situação em que seu browser não oferece um console JavaScript e você não pode instalar um novo, uma alternativa é aproveitar a barra de endereços como

prompt.

Vamos digitar no browser o seguinte endereço:

```
javascript:alert(readTotal());
```

Todo o conteúdo após os dois pontos serão enviados ao `engine` JavaScript do browser, que vai interpretá-lo e executá-lo, conforme a imagem 3.7.

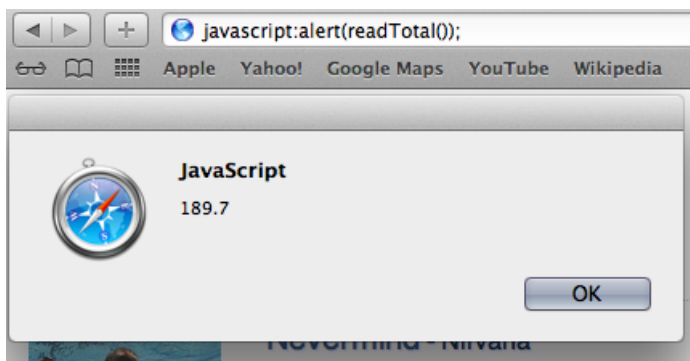



Figura 3.7: Resultado do comando via barra de endereços

Não é tão bom quanto um console, mas ainda é melhor do que a prática de espalhar `alert`s pelo código.

3.10 Calculando os subtotais dos itens

Para calcular o subtotal de cada produto, vamos precisar da quantidade e do valor unitário de cada produto que estiver no carrinho de compras.

Uma boa abordagem para fazermos isso é ler produto a produto do carrinho, calcular o subtotal e armazenar esse valor para que possamos utilizá-lo no cálculo do valor total.

Para podermos saber o que é um produto dentro de nossa página, vamos voltar ao nosso código HTML e adicionar a classe `produto` a cada linha da tabela que contém o nosso carrinho de compras.

Precisamos também demarcar onde podemos encontrar o valor unitário e a quantidade a ser comprada pelo usuário. Para isso vamos adicionar, respectivamente, as classes `price` e `quantity`.

Teríamos algo assim:

```
<table>
  <tbody>
    <tr class="produto">
      <td>
        <div class="price">R$ 29,90</div>
      </td>
      <td>
        <input type="number" class="quantity">
      </td>
    </tr>
    <!-- um produto de 59,90 -->
    <!-- um produto de 99,90 -->
  </tbody>
  <tr>
    <td></td>
    <td>Total da compra</td>
    <td><div id="total">R$ 189,70</div></td>
    <td></td>
  </tr>
</table>
```

Vamos utilizar um método irmão do `getElementById` para nos retornar os elementos que contêm a classe `produto`. Como vários elementos podem ter a mesma classe, o método `getElementsByClassName` vai nos retornar um `array` de elementos.

```
var produtos = document.getElementsByClassName("produto");

console.log(produtos);
=> [<tr class="produto">...</tr>]
```

Vamos procurar o elemento com a classe `price` dentro de cada um dos itens do `array`. Como nesse caso `getElementsByClassName` vai nos retornar um `array` de uma posição, podemos acessar diretamente a primeira posição sem maiores preocupações.

Com a tag em mãos, vamos extrair o valor unitário do produto com a função `moneyTextToFloat` e exibi-lo no console.

Console

No *Chrome*, *Firebug* e *Opera*, você pode adicionar mensagens de depuração no código usando `console.debug(mensagem)`.

No *Internet Explorer* e no *Firefox* sem *Firebug*, o comando vai causar um erro e parar a execução do seu script.

Com isso, nosso código vai ficar assim:

```
for(var pos = 0; pos < produtos.length; pos++) {
  var priceElements = produtos[pos].getElementsByClassName("price");
  var priceText = priceElements[0].innerHTML;
  var price = moneyTextToFloat(priceText);

  console.log(price);
}
```

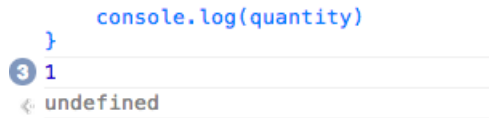
Veremos a listagem de preços, como na figura 3.8.

```
    console.log(moneyTextToFloat(priceText));
}
29.9
59.9
99.9
undefined
```

Figura 3.8: Totais listados no console

Vamos então listar as quantidades, dentro do mesmo `for` que acabamos de escrever:

```
var qtyElements = produtos[pos].getElementsByClassName("quantity");
var qtyText = qtyElements[0].value;
var quantity = moneyTextToFloat(qtyText);

console.log(quantity);
```

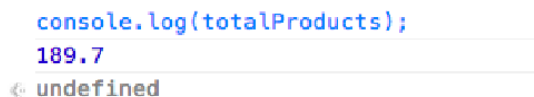
Agora vemos a quantidade dos itens na figura 3.9. Esse **3** dentro de um círculo indica que o valor foi repetido três vezes. O console evitou imprimir o valor repetidas vezes, já que isso ajuda a manter a saída legível.

```
    console.log(quantity)
}
③ 1
undefined
```

Figura 3.9: Quantidades listadas no console

Temos então a quantidade e o preço unitário de cada produto. Multiplicamos um pelo outro e somamos a uma variável que vai nos retornar o valor total dos produtos.

Precisamos declarar a variável `totalProdutos` antes do `for`, zerando o valor antes de começarmos a acumular.

```
var totalProdutos = 0;

for(var pos = 0; pos < produtos.length; pos++) {
// e o código continua.
```

Finalmente somaremos os valores para obtermos o total:

```
var subtotal = quantity * price;
  totalProdutos += subtotal;
}
```

Após o fechamento do `for`, vamos exibir o total que calculamos:

```
console.log(totalProdutos);
```

Eis que agora, na figura 3.10, temos o valor calculado de todos os produtos do nosso carrinho.

```
console.log(totalProducts);
189.7
undefined
```

Figura 3.10: Total dos produtos do carrinho

Para ficar bom mesmo, vamos encapsular esse código todo dentro de uma função, que chamaremos de `calculateTotalProducts`. O nosso código, depois de todo o trabalho, vai ficar assim:

```
function calculateTotalProducts() {
  var produtos = document.getElementsByClassName("produto");

  var totalProdutos = 0;

  for(var pos = 0; pos < produtos.length; pos++) {
    var priceElements = produtos[pos].
                getElementsByClassName("price");
    var priceText = priceElements[0].innerHTML;
    var price = moneyTextToFloat(priceText);

    var qtyElements = produtos[pos].
                getElementsByClassName("quantity");
    var qtyText = qtyElements[0].value;
    var quantity = moneyTextToFloat(qtyText);

    var subtotal = quantity * price;

    totalProdutos += subtotal;
  }

  return totalProdutos;
}
```

Fique tranquilo! Códigos longos como esse serão uma exceção por aqui.

Onde encontrar os códigos do livro

Os códigos do livro podem ser encontrados no endereço abaixo, separados por capítulos:

https://github.com/pbalduino/livro-js-jquery

Fique à vontade para alterá-los da forma que achar melhor.

3.11 Entendendo eventos

Eventos são chamadas de código que ocorrem quando o usuário ou o browser executam determinadas ações. Temos eventos para quando o usuário clica em algum lugar, para quando move o ponteiro do mouse sobre uma região, ou para quando o ponteiro do mouse sai dessa região.

Aprender a trabalhar com eventos é parte importante do desenvolvimento com JavaScript.

Vamos adicionar um código que atualiza o valor total do carrinho de compras sempre que o usuário modificar a quantidade de um produto, sem que para isso ele tenha que clicar em algum botão.

Começaremos criando a função que deverá ser chamada toda vez que alguém modificar uma quantidade:

```
function quantidadeMudou() {
  writeTotal(calculateTotalProducts());
}
```

Mas como fazer com que essa função seja invocada? Basta pegarmos todos os campos que têm a classe `quantity` e falar que, sempre que houver uma mudança (`onchange`), a função `quantidadeMudou` deve ser chamada:

```
function onDocumentLoad() {
  var textEdits = document.getElementsByClassName("quantity");

  for(var i = 0; i < textEdits.length; i++) {
    textEdits[i].onchange = quantidadeMudou;
  }
}
```

Pronto! Toda vez que houver uma mudança nesse campo de edição, a nossa função `quantidadeMudou` será executada.

Mas quem vai chamar essa função `onDocumentLoad` que acabamos de criar? Basta dizer que ela deve ser executada assim que o evento do carregamento da janela acontecer! Para isso fazemos:

```
window.onload = onDocumentLoad;
```

Para testar, atualize a página e altere o valor de qualquer um dos itens.

3.12 A seguir, cenas do próximo capítulo

Nosso código ficou bacana e funciona redondinho no Chrome e no Safari. Mas e nos outros browsers?

No Internet Explorer, por exemplo, ele não vai nem ser interpretado até o final.

No próximo capítulo estudaremos as particularidades dos principais browsers e vamos mostrar como escrever códigos que sejam compatíveis com eles.

Capítulo 4

Um JavaScript diferente em cada navegador

"Anyone who slaps a 'this page is best viewed with Browser X' label on a Web page appears to be yearning for the bad old days, before the Web, when you had very little chance of reading a document written on another computer, another word processor, or another network."
– Tim Berners-Lee

4.1 Nem tudo são flores no reino da Web

Nosso carrinho de compras utiliza uma pesquisa por classes para calcular o total do pedido, e atualiza automaticamente esse valor sempre que o usuário alterar a quantidade de produtos.

Porém, essa pesquisa por classes simplesmente não funciona no Internet Explorer 8 e versões anteriores. Isso se deve ao fato de que apenas recentemente a Microsoft

começou a se preocupar com os padrões da Web, fazendo com que tenhamos que escrever código extra para que nosso site funcione no Internet Explorer.

Limitações dos browsers

Para descobrir essas e outras limitações, você pode usar o site *Can I Use...*:

http://caniuse.com/

Lá você pode ver que a função `getElementsByClassName` só foi implementada nesse navegador a partir da versão 9.

O que fazer agora? Abandonar essa versão do navegador e perder possíveis clientes que usam uma versão mais antiga? Utilizar outra abordagem?

A solução vai ser escrever nosso próprio código que faça a análise de todos os elementos do nosso documento e retorne somente aqueles que pertençam à classe que queremos. Sim, vai dar trabalho. De qualquer maneira podemos aproveitar os navegadores que já possuem a função `getElementsByClassName` em vez de criar uma nova.

No próximo capítulo vamos demonstrar como passar por cima da incompatibilidade que existe entre os diferentes navegadores e suas versões. Perceba que isso ocorre mesmo para as funções mais básicas, para as necessidades mais simples do nosso website.

4.2 Quando não existe uma determinada função

Vamos criar nosso próprio código [11] que faça essa seleção no Internet Explorer inferior à versão 9. Em muitas linguagens precisaríamos criar uma função e, fazendo uso de um `if`, escolher qual delas utilizar. Felizmente, com JavaScript você consegue detectar em tempo de execução se aquela função está disponível ou não e, se necessário, criar uma função com o mesmo nome daquela que está faltando. Para isso, vamos executar o código abaixo:

```
alert(document.getElementsByClassName === undefined);
```

Veremos a mensagem no Internet Explorer, conforme a figura 4.1:

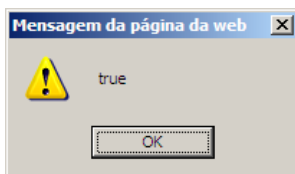



Figura 4.1: IE não conhece o getElementsByClassName

Sempre que você verificar uma função ou uma variável que não existe, você receberá o valor `undefined` como resposta. Perceba que não usamos `()` após `getElementsByClassName`, pois isso faz com que a função seja executada, o que vai causar um erro caso ela não exista.

```
if (document.getElementsByClassName == undefined) {
  alert("getElementsByClassName not found");
}
```

Dentro desse `if`, vamos criar o método `getElementsByClassName` para o objeto `document`. Para isso, basta atribuirmos uma *função anônima* ao nome do método. Essa função precisa receber o nome da classe como parâmetro, como acontece com os outros browsers.

```
document.getElementsByClassName = function(className) {
  alert("Regozijai-vos, usuários de Internet Explorer");
}
```

Vamos então usar o método `getElementsByTagName` com o caractere coringa asterisco para criar um `array` com todos os elementos do nosso documento. Em seguida, vamos verificar cada um desses itens e separar em outro `array` somente aqueles que tenham a classe que estamos procurando.

Como uma boa prática, quando você precisar acessar mais de uma vez seguida o mesmo valor de uma propriedade ou método, jogue esse valor numa variável local e passe a usá-la, como vamos fazer com `elementClass`. O JavaScript consome um bom tempo localizando propriedades dentro de um objeto e isso nos faz ganhar alguma performance.

Declaração de variáveis dentro ou fora de um for

Existem algumas discussões em fóruns especializados e grupos de usuários sobre ser melhor declarar uma variável dentro ou fora de um `for` ou qualquer outro tipo de iteração.

Se, por um lado, existem argumentos dizendo que a declaração da variável a cada iteração torna o processo mais lento, existem outros em igual quantidade dizendo que o fato da variável não estar no menor escopo possível também causa perda de performance.

No final das contas, efetuando testes em diferentes browsers e sistemas operacionais, não notei quaisquer diferenças significativas de performance e de tempo de execução. Como *a otimização prematura é a raiz de todos os males*, use a forma que deixar seu código mais legível e expressivo.

Se o elemento que está sendo verificado no `for` não tiver uma classe, `elementClass` será `undefined` e funcionará como `false` dentro do `if`. Ou seja, `if(elementClass)` e `if(elementClass !== undefined)` são equivalentes.

```
var todosElementos = document.getElementsByTagName("*");
var resultados = [];

var elemento;
for (var i = 0; (elemento = todosElementos[i]) != null; i++) {
  var elementoClass = elemento.className;
  if (elementoClass &&
      elementoClass.indexOf(className) != -1) {
    resultados.push(elemento);
  }
}
return resultados;
```

Para o que precisamos fazer na nossa loja virtual, esse código é mais do que suficiente, mas ele apresenta um problema que vou deixar para você resolver, como forma de exercício: o que acontece se eu pesquisar elementos que tenham duas classes simultaneamente, como no código abaixo?

```
<span class="message alert">Lembre-se que a tag span não é spam</span>
```

E se eu tiver dois elementos com as mesmas duas classes declaradas em ordem diferente? Considere a seguinte listagem de bandas de rock:

```
<span class="product hot">Screaming Trees</span>

<span class="hot product">Sweet Oblivion</span>
```

A implementação de `getElementsByClassName` retorna os dois elementos `span`, independentemente da ordem declarada. Seu exercício é melhorar a nossa implementação para que ela possa fazer o mesmo.

4.3 Funções anônimas e nomeadas

Existem dois tipos de funções no JavaScript: *anônimas* e *nomeadas*.

As funções nomeadas são as que já conhecemos, e são invocadas através do próprio nome:

```
function olaMundo() {
  console.log("Ola");
}

olaMundo();
```

Já as funções anônimas não têm um nome para serem invocadas diretamente, mas são muito usadas como parâmetros para outras funções, ou são atribuídas a uma variável, que acaba funcionando como um nome.

```
function() {
  console.log("Nunca serei executado :'( ")
}

var olaMundo = function() {
  console.log("Serei executado =D");
}

olaMundo();
```

Passaremos muitas funções anônimas como argumento, como já veremos com o jQuery. Em muitos casos eles são os callbacks, aquelas funções que são chamadas quando queremos ser informados de que algo ocorreu.

Vamos rever como fizemos o callback da seção anterior. Primeiro declaramos a função `quantidadeMudou` e depois, dentro de uma outra função, definimos que todos os eventos `onChange` devem chamar o callback `quantidadeMudou`:

```
function quantidadeMudou() {
  writeTotal(calculateTotalProducts());
}

function onDocumentLoad() {
  var textEdits = document.getElementsByClassName("quantity");

  for(var i = 0; i < texts.length; i++) {
    textEdits[i].onchange = quantidadeMudou;
  }
}
```

Aqui utilizamos uma função com nome bem explícito. Mas em JavaScript, quando uma função só é chamada uma vez e ela é relativamente simples e curta, é frequente não declará-la dessa forma. Fazemos tudo junto, de forma anônima. Repare:

```
function onDocumentLoad() {
  var textEdits = document.getElementsByClassName("quantity");

  for(var i = 0; i < texts.length; i++) {
    textEdits[i].onchange = function() {
      writeTotal(calculateTotalProducts());
    };
  }
}
```

Ficou menor, mas em algumas vezes pode complicar a leitura. Fica a decisão do programador, mas habitue-se a ler código dessa forma.

Capítulo 5

Simplifique com jQuery

"Não invente. Faça o básico!"
– Professor Lúcio Antonio dos Santos

Resolver esses nossos pequenos problemas que aparecem nos diferentes navegadores não seria fácil. São diversos Internet Explorers, Safaris, Firefoxes e Chromes, além de outros menores. Adivinhe? Alguém já quebrou a cabeça com um monte de ifs e casos particulares e criou uma biblioteca cheia de comandos para resolver os casos mais comuns no desenvolvimento web. O jQuery é sem dúvida o líder desse tipo de biblioteca.

5.1 O que é jQuery

Numa tradução livre, de acordo com o próprio site da ferramenta, jQuery é uma biblioteca JavaScript que simplifica a manipulação de documentos HTML, eventos, animações e interações com AJAX para desenvolvimento rápido de aplicações web.

Devido à sua simplicidade e flexibilidade, o jQuery acabou se tornando a biblioteca JavaScript mais utilizada[17]. Em 2008 passou a ser a biblioteca padrão do ASP.NET MVC, da Microsoft, e o Ruby on Rails, a partir da versão 3, substituiu o Prototype pelo jQuery.

A maneira mais simples de utilizar o jQuery é fazendo o download do arquivo JavaScript no endereço:

http://jquery.com/download/

A menos que você pretenda alterar ou estudar o código do jQuery, selecione a opção *production/minified*, que é compactada e faz com que seu site leve menos tempo para ser aberto.

5.2 Qual versão do jQuery usar? 2.0 ou 1.x

Em abril de 2013, foi lançando o jQuery 2.0. Ele é totalmente compatível com a versão 1.9, isto é, você pode chamar todas as funções do 1.9 mesmo se estiver usando o 2.0.

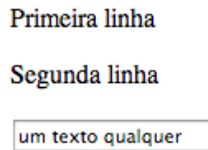
Obviamente o 2.0 tem uma série de vantagens, como um tamanho menor, mais customização para pegar só que você precisa da biblioteca e muitos novos recursos.

Há também uma grande desvantagem. O jQuery 2.0 não dá mais suporte ao Internet Explorer 6, 7 e 8. Esses navegadores estão numa queda acentuada de uso mas, dependendo do site, pode ser importante mantê-los compatíveis. Cabe a você a decisão.

Você pode ler mais aqui: http://blog.jquery.com/2013/04/18/jquery-2-0-released/

Focamos o livro em utilizar chamadas que funcionam tanto no 1.x quanto no 2.x, tendo em vista que o 1.x vai continuar, por muito tempo, sendo o mais utilizado.

5.3 Entenda jQuery em cinco minutos

Uma expressão jQuery é formada de duas partes principais: **o que** vai ser manipulado e **como** isso vai acontecer.

Vamos considerar que nossa página tenha o trecho abaixo, com dois parágrafos e uma caixa de texto. O exemplo a seguir está disponível no endereço:

http://jsfiddle.net/pbalduino/MVAY3/

Crie um arquivo `primeiro_jquery.html` com o seguinte conteúdo:

```
<p class="par">Primeira linha</p>
<p class="impar">Segunda linha</p>
<input type="text" id="texto" class="par" value="um texto qualquer">
```

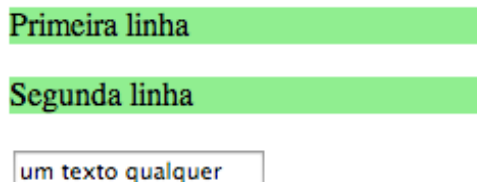
Abra o HTML no seu navegador. Vai ficar com a cara da imagem 5.1.

Primeira linha

Segunda linha

um texto qualquer

Figura 5.1: Nossa página antes de usarmos o jQuery

No primeiro exemplo, vamos **selecionar** todos os parágrafos, que são os elementos com a tag p, e mudar a cor de fundo para verde. Você pode executar esses exemplos no console do seu browser se já conhecer um pouco mais. Se não, edite o arquivo primeiro_jquery.html e adicione ao final dele:

```
<script src="http://code.jquery.com/jquery-2.0.0.js"></script>

<script>

$("p").css("background-color", "lightgreen");

</script>
```

A primeira linha carrega o jQuery (vamos falar de boas práticas mais a frente). A linha dentro do script faz exatamente o que queríamos: seleciona todos os elementos p e muda um atributo do css. Temos o resultado na imagem 5.2.

Figura 5.2: Depois de selecionarmos e alterarmos a tag ‘p

Simples, não? A linha `$("p").css("background-color", "lightgreen");` é bem legível: selecione os `p`s e mude `background-color` para `lightgreen` de todos eles!

O que significa o $, afinal?

O caractere `$` é usado como um modo bem curto de dizer `jQuery`. Sempre que você ver `$("p")`, saiba que isso faz exatamente a mesma coisa que `jQuery("p")`. Então, leia o `$` como `jQuery` e tudo bem.

Agora vamos selecionar todos os elementos que contenham a classe `par` e vamos mudar sua cor de fundo para azul. Edite novamente seu arquivo e adicione:

```
$(".par").css("background-color", "lightblue");
```

E vemos como ficou na imagem 5.3.

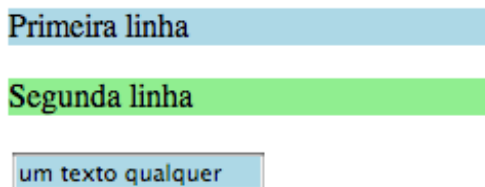



Figura 5.3: Depois de selecionarmos e alterarmos a classe 'par'

Repare que **selecionamos** elementos pela classe ou pela tag HTML utilizando a mesma forma `$("alguma_coisa")`, mas no caso da classe utilizamos um ponto na frente.

E adivinhe? Podemos também **selecionar** um elemento também através do ID, bastando utilizar o `#` na frente da string. Para exemplificar, vamos alterar a cor da fonte e o conteúdo da nossa caixa de texto:

```
$("#texto").css("color", "white").val("esse é o novo texto");
```

Note que agora, após selecionarmos o elemento com `.css(...)`, usamos duas funções juntas para alterar a cor da fonte e o texto. A isso chamamos *encadeamento*

de funções, e é uma das principais características do jQuery: a função `css` retorna o próprio elemento já com a alteração que fizemos, e assim você pode continuar alterando outros valores, como estamos fazendo com o `val`, que altera o valor de um campo de `input`.

A imagem 5.4 mostra como nossa página fica depois de das alterações:

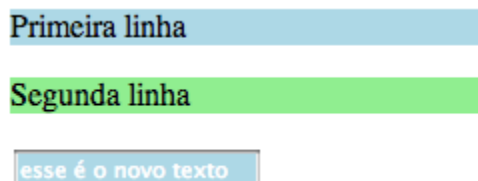



Figura 5.4: Depois de alterarmos a cor da fonte e o texto usando o id 'texto'

Nos exemplos que vimos, selecionamos os elementos usando as três formas mais comuns: pesquisando pela tag HTML, simplesmente informando a tag; pela classe, usando um ponto final antes do nome da classe; e pelo ID, usando o caractere `#` antes do identificador.

Sim, esses são os famosos selectors!

Essas formas de selecionar os elementos são chamados **selectors**, e são parte fundamental do jQuery. Os **selectors** indicam **o quê** nós vamos manipular. Basicamente, entendendo o conceito de **selectors**, você entende a maior parte do funcionamento do framework.

A melhor parte dos exemplos que executamos é que eles rodam sem alterações em qualquer browser moderno. Sabendo disso, vamos ver como ficaria o nosso carrinho de compras usando jQuery?

5.4 Nosso código antigo, agora com jQuery

Agora que você entendeu como usar o jQuery, vamos reescrever nosso carrinho de compras usando a biblioteca.

O primeiro ponto que você vai notar, conforme já repetimos aqui, é que você não vai precisar de um código específico para o Internet Explorer. Os desenvolvedores do jQuery já fizeram o trabalho pesado para nós.

Vamos seguir a mesma ordem do capítulo 3, demonstrando como podemos retornar o valor total do carrinho de compras:

Lembre-se que vamos editar o arquivo `javascripts/rogus.js` para adicionar os códigos a seguir.

```
function readTotal() {
  var total = $("#total").text();
  return moneyTextToFloat(total);
}
```

Assim como o `val` lê e altera a propriedade `value` de um elemento `input`, a função `text` trabalha com o texto que está dentro das outras tags, ignorando formatação e quaisquer outras tags que estejam dentro do elemento.

Vamos reescrever a função que altera o valor total. Note que o método `text` serve tanto para ler quanto para escrever o novo texto dentro do elemento `total`. O jQuery usa muito desse artifício para trabalhar com propriedades.

```
function writeTotal(value) {
  var text = floatToMoneyText(value);
  $("#total").text(text);
}
```

Intuitivo, não?

Sabendo como ler o valor de um `input` e como ler o texto das outras tags, vamos reescrever a função que calcula o valor total dos itens.

Primeiro, vamos selecionar os elementos com a tag `produto`, que contém a imagem do produto, descrição, preço unitário e quantidade. A variável `produtos` vai conter um elemento para cada produto do carrinho de compras.

```
var produtos = $(".produto");
```

Dentro de cada produto do carrinho, precisamos pegar o preço unitário e a quantidade. Para encontrarmos o valor de um elemento (preço ou quantidade) que está dentro de outro elemento (produto do carrinho de compras), precisamos usar o método `find`.

Vamos tentar ler a quantidade do primeiro produto do carrinho.

```
var primeiroProduto = produtos[0];

primeiroProduto.find(".quantity").val();
```

Hmm... algo deu errado, conforme podemos ver na imagem 5.5.

```
> var primeiroItem = items[0];
  undefined
> primeiroItem.find(".quantity").val();
✖ ▶ TypeError: Object #<HTMLTableRowElement> has no method 'find'
> |
```

Figura 5.5: Algo não funcionou como esperado

O que acontece aqui é que, assim como o método `document.getElementsByClass`, a instrução `$(".produto")` retornou um `Array` de **elementos do DOM**, e não de **objetos jQuery**, e o método `find` só existe para objetos jQuery. Para resolver isso, basta colocar o elemento dentro do `$()`, e ele é convertido para um objeto jQuery, permitindo que possamos usar todas as funcionalidades da biblioteca.

```
var primeiroProduto = produtos[0];

$(primeiroProduto).find(".quantity").val();
```

Na imagem 5.6 vemos a quantidade a ser comprada do primeiro produto do carrinho.

```
> var primeiroItem = items[0];
  undefined
> $(primeiroItem).find(".quantity").val();
  "1"
```

Figura 5.6: Agora funcionou

Agora que estamos atentos a essa sutileza da biblioteca, podemos reescrever a função que calcula o total do pedido usando a mesma abordagem do capítulo anterior, mas de modo mais conciso.

```
function calculateTotalProducts() {
  var produtos = $(".produto");
  var total = 0;
```

```
  for(var pos = 0; pos < produtos.length; pos++) {
    var $produto = $(produtos[pos]);
    var quantity = moneyTextToFloat(
          $produto.find(".quantity").val());
    var price = moneyTextToFloat(
          $produto.find(".price").text());
    total += quantity * price;
  }
  return total;
}
```

Compare com o código que tínhamos anteriormente e surpreenda-se.

5.5 Boas práticas e convenções com jQuery

Às vezes pode ficar complicado descobrir qual variável é um elemento DOM e qual é um elemento jQuery. Apesar de não ser uma convenção aceita por todo desenvolvedor, algumas pessoas indicam usar o `$` na frente da variável que for do jQuery. Nesse caso nosso código ficaria `var $produto = $(produtos[pos]);`. Dessa forma, toda vez que você lesse no código `$produto`, saberia que se trata de um elemento jQuery.

Há mais sugestões de convenções aqui: http://www.jameswiseman.com/blog/2010/04/20/jquery-standards-and-best-practice/

E aqui um argumento contra essa prática: http://bears-eat-beets.blogspot.com.br/2011/08/jquery-naming-conventions-dont-prefix.html

Nós usaremos o prefixo `$` no livro para auxiliar o leitor a entender melhor o código, mas fique a vontade para decidir o que for melhor para você ou para sua equipe.

5.6 Programando de forma funcional

Apesar do nosso `for` resolver bem o problema, é bastante comum um desenvolvedor jQuery utilizar uma abordagem um pouco mais funcional. Em vez de fazer uma iteração por um loop, vamos passar uma função como argumento para alguém que a executará para cada um dos itens. Essa é a função `each`:

```
function calculateTotalProducts() {
  var produtos = $(".produto");
```

```
  var total = 0;

  $(produtos).each(function(pos, produto) {
    var $produto = $(produto);
    var quantity = moneyTextToFloat(
                   $produto.find(".quantity").val());
    var price = moneyTextToFloat(
                   $produto.find(".price").text());

    total += quantity * price;
  });

  return total;
}
```

Essa `function()` que passamos como argumento para o `each` será invocada para cada um dos nossos itens do carrinho. Você já deve estar cansado de saber, mas é muito comum passar funções como argumento para outras funções. Fazemos isso o tempo todo, em especial com os *callbacks* (exemplo: clicou em um botão de excluir produto, chame a função `compraAtualizada()`). O nome técnico disso até assusta, são as chamadas *high order functions*.

Mas como saber o que a função `each` recebe como argumento? Assim como o JavaScript possui uma excelente documentação no site da Mozilla, o jQuery tem sua própria documentação! Repare:

http://api.jquery.com/each/

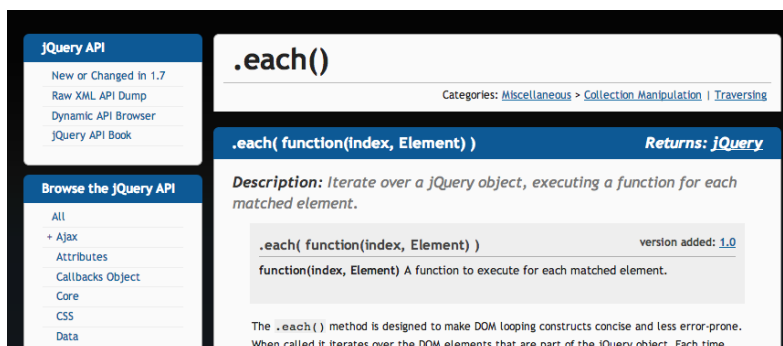



Figura 5.7: Documentação do each de um selector jQuery

É aí que você fica sabendo que o `each` recebe uma função. Além disso, essa função receberá como parâmetro um `index` e um `Element`, que representam, respectivamente, a posição que o elemento ocupa e o elemento em si. No nosso caso, o `Element` é o produto atual da iteração.

Navegadores que suportam o padrão ECMAScript mais moderno já permitem que você chame a função `each` em uma array, além de implementarem diversas outras interessantes funcionalidades. Mas, como já aprendemos, diversos navegadores ainda não possuem esses recursos mais recentes e por isso ainda é interessante utilizá-los através de uma biblioteca como o jQuery.

5.7 Eventos e callbacks

Assim como acontece com a manipulação do DOM, o jQuery também nos auxilia no tratamento de eventos. Recebemos as chamadas informando desses eventos em *callbacks*, através de funções que são passadas como parâmetro.

O primeiro ponto que vamos notar de diferente é que não precisamos definir os eventos de cada um dos elementos, como fizemos usando JavaScript puro. Basta selecionarmos a classe que queremos e o jQuery se encarrega de definir o evento para todos os elementos que forem encontrados.

Para definirmos que, cada vez que um campo da classe `quantity` for alterado, o total do carrinho seja recalculado, nós vamos passar uma função como parâmetro para o método `change`. Toda vez que o campo for alterado, essa função será executada e o total do nosso carrinho será recalculado.

```
$(".quantity").change(function() {
  writeTotal(calculateTotalProducts());
});
```

Compare isso com a versão anterior do código! Olhe só:

```
function onDocumentLoad() {
  var textEdits = document.getElementsByClassName("quantity");

  for(var i = 0; i < texts.length; i++) {
    textEdits[i].onchange = function() {
      writeTotal(calculateTotalProducts());
    };
  }
}
```

A versão do jQuery é sem dúvida mais sucinta e até mais fácil de ler. Sem contar a vantagem de trabalhar da mesma forma em diferentes navegadores.

Agora, para definirmos esse evento assim que o documento terminar de ser carregado, vamos usar outra funcionalidade muito importante do jQuery.

O jQuery tem um evento chamado `ready`, que é executado assim que o documento terminar de ser carregado. O funcionamento é igual a qualquer outro evento: passamos uma função como parâmetro, que será executada assim que o evento for disparado.

Nosso código então ficaria assim:

```
jQuery(document).ready(function() {
  // faça alguma coisa
});
```

Porém, você deve se lembrar que eu disse que `jQuery` e `$` são equivalentes. Então podemos substituir um pelo outro, deixando o código ainda menor:

```
$(document).ready(function() {
  // faça alguma coisa
});
```

Como se isso não bastasse, os desenvolvedores do jQuery foram preguiçosos o bastante para deixar essa expressão menor ainda. Você pode simplesmente omitir o `ready`, passando uma função como parâmetro diretamente para o `$`. Com isso, usando a nossa declaração de evento lá de cima, nosso código finalmente vai ficar com essa cara:

```
$(function() {
  $(".quantity").change(function() {
    writeTotal(calculateTotalProducts());
  });
});
```

E pronto! Temos o nosso carrinho de compras funcionando em todos os navegadores modernos usando muito menos código. Claro, se seu código aí dentro for grande, era melhor quebrar em uma função que tenha um nome expressivo.

5.8 E o que vem agora?

O que vimos até agora serviu para nos mostrar as vantagens do jQuery e o quanto ele agiliza o desenvolvimento do nosso código, mas ainda não é o bastante. Nos pró-

ximos capítulos vamos ver como manipular o DOM, como calcular o frete usando AJAX e como adicionar efeitos visuais interessantes no nosso carrinho de compras. O melhor de tudo é que você pode ficar tranquilo que isso vai rodar em qualquer browser moderno.

Capítulo 6

Dominando eventos e manipulação de DOM com jQuery

6.1 Criando uma lista de tarefas

Para aprendermos mais a fundo como manipular o DOM usando jQuery, vamos desenvolver uma aplicação que gerencie nossas tarefas cotidianas.

Relembro que você pode encontrar o código desse capítulo e de outros no repositório do livro:

https://github.com/pbalduino/livro-js-jquery/tree/master/capitulo_06

Mas escrever do zero sempre agrega mais ao aprendizado, reforçando alguns conceitos já vistos na prática.

A nossa aplicação vai ter uma caixa de texto onde podemos digitar uma nova tarefa e logo abaixo a lista de tarefas que já incluímos. A ideia é termos a interface mais limpa e simples possível para focarmos no código que vamos escrever.

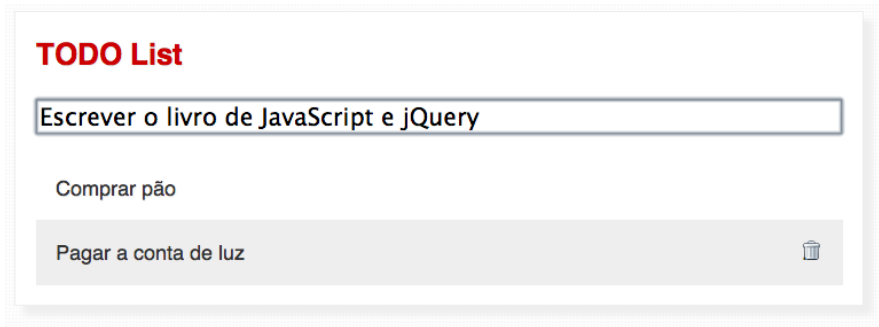



Figura 6.1: A cara da nossa lista de tarefas

O código HTML da nossa lista terá um elemento `input` com id `tarefa`, que vamos usar para inserir as tarefas e teremos um elemento `div` com id `tarefa-lista`, que vai conter todos as tarefas.

Cada tarefa vai ficar dentro de um outro elemento `div` que terá a classe `tarefa-item`, que terá mais três elementos `div`: um que contém a descrição da tarefa, que terá a classe `tarefa-texto`; outro elemento `div` que servirá para excluir a tarefa quando for clicado, que vai ter a classe `tarefa-delete`; por fim, apenas para fins de formatação, teremos um terceiro elemento `div` que conterá a classe `clear`.

Listagem 6.1 -

:

```
<h2>TODO List</h2>
<input type="text" id="tarefa">
<div id="tarefa-lista">
  <div class="tarefa-item">
    <div class="tarefa-texto">Comprar pão</div>
    <div class="tarefa-delete"></div>
    <div class="clear"></div>
  </div>
</div>
```

Com esses elementos definidos, conforme o código 1, vamos começar a desenvolver nossa lista de tarefas.

6.2 Usando eventos de um jeito mais profissional

Para adicionar novas tarefas, o usuário precisa digitar um texto e pressionar *ENTER*. Para que possamos detectar o uso dessa tecla, vamos usar o evento `keydown`.

Note que esse método recebe um objeto contendo as informações do evento. Isso vai ser muito útil para descobrirmos que tecla está sendo pressionada. Esse objeto tem uma propriedade `which`, que retorna o código Unicode da tecla usada.

Vamos brincar um pouco com esse evento para ver como ele se comporta. Você pode fazer esses testes direto no console do seu navegador:

```
$('#tarefa').keydown(function(event) {
  console.log(event.which, String.fromCharCode(event.which));
});
```

Ao escrevermos *evento* e pressionar ENTER, veremos a saída da figura 6.2. O método `String.fromCharCode`, quando possível, converte o código da tecla pressionada, que é um número, para o respectivo caractere, tornando o nosso exemplo mais legível.

```
69 "E"
86 "V"
69 "E"
78 "N"
84 "T"
79 "O"
13 ""
>
```

Figura 6.2: Key down

O `event.which` das letras é o respectivo código `Unicode`[14], mas no ENTER recebemos o valor `13`. Sabendo disso, vamos fazer o nosso exemplo trabalhar apenas quando for detectado que o usuário pressionou essa tecla, digitando o código abaixo no console do browser.

Listagem 6.2 -

:

```
$('#tarefa').keydown(function(event) {
  if(event.which === 13) {
    console.log("Aqui vamos adicionar nossa tarefa");
  }
});
```

Digitando o mesmo texto "*evento*" e pressionando ENTER, teremos a saída abaixo:

```
69 "E"
86 "V"
69 "E"
78 "N"
84 "T"
79 "O"
13 ""
Aqui vamos adicionar nossa tarefa
>
```

Figura 6.3: Key down com dois eventos

Mas não estávamos esperando que, ao disparar o evento, o jQuery executasse as duas funções que escrevemos. Isso permite que você atribua funções de callback diferentes para o mesmo evento sem que uma sobrescreva a outra. Imagine como seria ruim se você definisse uma função para esse evento `keydown` e, na sequência um código feito por outra pessoa simplesmente apagasse o que você fez para executar o código dele.

Caso você queira **remover** as funções que estão associadas a um evento, você deve usar o método `off`, passando o nome do evento como parâmetro. Você pode usar `$("#tarefa").off("keydown");` nesse caso. Há também a função `off`, mais recente, recomendada a ser usada a partir do jQuery 1.7.

Se executarmos novamente o código da listagem 3, veremos a saída exibida na figura 6.4

Listagem 6.3 -

:

```
$('#tarefa').off();
```

```
$('#tarefa').keydown(function(event) {
  if(event.which === 13) {
    console.log("Aqui vamos adicionar nossa tarefa");
  }
});
```

```
Aqui vamos adicionar nossa tarefa
> |
```

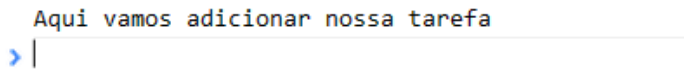



Figura 6.4: Key down novamente com um evento

6.3 Desassociando eventos

Pode acontecer de um código de terceiros atribuir um evento para um elemento e você precisar atribuir o seu próprio código para o mesmo evento do mesmo elemento. Como já vimos, o jQuery vai executar os dois em sequência, sem problema algum.

A situação começa a ficar ruim quando você ou o código de terceiros precisam remover somente um dos códigos do evento. Como fazer isso?

Vamos ver na prática como funciona, usando o mesmo código HTML que já temos pronto. Para isso, atualize sua página pressionando `F5` ou `Command R` e vamos usar novamente o console do jQuery:

```
$("#tarefa").keydown(function() {
  console.log("Esse é o primeiro evento");
});

$("#tarefa").keydown(function() {
 console.log("Esse é o segundo evento");
});
```

Ao pressionar alguma coisa tecla, veremos as duas mensagens exibidas na figura 6.5.

```
Esse é o primeiro evento
Esse é o segundo evento
>
```

Figura 6.5: Eventos encadeados

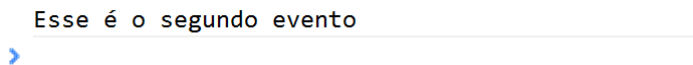
Se usarmos o `off`, tanto o primeiro quanto o segundo evento serão eliminados. Para resolvermos isso usamos algo chamado **namespaces**.

Para utilizar o **namespace** em um evento, vamos utilizar o método `on`, informar o evento que queremos declarar e o nome do **namespace**, separados por ponto, e também a função a ser executada quando o evento for disparado.

Aproveitando o exemplo que usamos anteriormente, atualize novamente a janela do seu browser e vamos digitar o código abaixo no console:

```
$("#tarefa").on("keydown.primeiro", function() {
 console.log("Esse é o primeiro evento");
});

$("#tarefa").on("keydown.segundo", function() {
 console.log("Esse é o segundo evento");
});
```

BIND E UNBIND

Se você já utilizou eventos com jQuery, ou encontrou algum código que fazia isso, deve ter visto os métodos `bind` e `unbind`.

A partir da versão 1.7 do jQuery, é recomendado que você use `on` no lugar de `bind` e `off` no lugar de `unbind` porque, como costuma acontecer, em algum momento no futuro os métodos antigos serão removidos da biblioteca.

A sintaxe de `on` e `off` é totalmente compatível com `bind` e `unbind`, o que facilita muito a atualização do código.

Ao pressionarmos qualquer tecla, veremos exatamente a mesma saída do exemplo anterior, mas a diferença é que agora podemos desassociar somente um dos eventos, conforme o exemplo a seguir:

```
$("#tarefa").off("keydown.primeiro");
```

Agora, ao pressionar alguma tecla, vamos ver apenas a mensagem do segundo evento, conforme a figura 6.6. Dessa forma, você pode associar e desassociar seus eventos conforme a necessidade sem se preocupar em quebrar o código de terceiros.

Esse é o segundo evento

Figura 6.6: Primeiro evento removido

6.4 Removendo itens com estilo

Apesar de parecer que estamos começando do final, o código que remove um item da nossa lista de tarefas é o mais simples, e ainda por cima usa tudo o que aprendemos sobre associação e desassociação de eventos.

Dentro do nosso projeto, vamos criar um diretório chamado `javascripts` e, dentro dele, um arquivo com o nome de `todo.js`. Vamos usar esse nome de arquivo pelo fato do nosso projeto ter originalmente o nome de TODO List, ou lista de coisas a fazer em inglês. Não existe uma padronização formal para nomes de arquivos JavaScript, então sinta-se à vontade para usar aquele que descrever melhor o propósito do seu código ou projeto.

O começo você já conhece. Vamos usar aquele método `$(document).ready()` para associarmos todos os eventos necessários assim que o documento estiver carregado.

Dentro desse método, vamos associar algum código ao evento `click` daquele elemento `div` que vai exibir o ícone da lixeira.

```
$(function() {

  function onTarefaDeleteClick() {
    console.log("Adeus mundo cruel!");
  }

  $(".tarefa-delete").click(onTarefaDeleteClick);
});
```

Aqui você pode perceber que criamos uma função nomeada para atribuir ao evento `click`. Eu poderia muito bem ter utilizado uma função anônima mas, conforme o nosso código for crescendo, vai ficar praticamente impossível nos encontrarmos dentro de funções anônimas declaradas dentro de funções anônimas.

A essa bagunça generalizada os desenvolvedores deram o nome de **Callback hell**[15] ou, numa tradução livre, *o inferno dos callbacks*. Você pode acessar o link abaixo para entender o que é o **Callback hell** e como fugir dele:

http://callbackhell.com/

Outro ponto que você deve ter percebido é o nome que dei para a função que será executada quando o evento `click` for disparado. Não existe um padrão oficial para nomear funções no jQuery, mas conforme me foi dito pela própria equipe de desenvolvimento da biblioteca, *adote um padrão e siga-o até o fim*.

O padrão adotado nesse livro para nomear funções que serão associadas a eventos, os nossos já conhecidos callbacks, é:

on<*identificador do elemento*>**<nome do evento>**

No caso do código acima, o elemento usa a classe `tarefa-delete` para ser selecionado, então vamos converter esse nome para o padrão JavaScript, que é usar a primeira letra da palavra em maiúscula e as demais em minúsculas, e ele passará a ser `TarefaDelete`. O nome do evento é `click`, portanto vamos mudar a primeira letra para maiúscula e usaremos `Click`. Juntando tudo isso teremos `onTarefaDeleteClick`.

Dessa forma, ao ler o nome da função, você automaticamente vai saber do que se trata e para que ela serve.

Quando você clica no ícone da lixeira, você quer que a linha toda seja excluída, e não apenas o ícone. Para isso, precisamos encontrar uma forma de selecionar o item da tarefa que contém aquela lixeira. Para essa tarefa, usaremos o método `parent`, que retorna o **elemento pai**, ou seja, o primeiro elemento que estiver contendo o nosso elemento atual e que respeite as regras do seletor que utilizarmos.

Olhando o exemplo vemos que é bem mais simples do que parece:

```
function onTarefaDeleteClick() {
  console.log($(this).parent('.tarefa-item').text().trim());
}
```

E veremos o texto que está contido na nossa tarefa, indicando que selecionamos toda a linha. O `trim` no final da sentença está ali apenas para eliminar os espaços

em branco no começo e no fim do texto, tornando a exibição mais legível, como podemos ver na figura 6.7.

```
Comprar pão
> |
```

Figura 6.7: Comprar pão. Boa ideia.

Vamos usar o método `hide` para esconder o elemento. Esse método esconde, mas não exclui o elemento do DOM.

```
function onTarefaDeleteClick() {
  $(this).parent('.tarefa-item').hide();
}
```

O problema aqui é que o item sumiu de repente, mas ele continua fazendo parte do DOM. Usar o método `remove` teria o mesmo efeito visual e ainda por cima eliminaria o elemento de uma vez. A vantagem do `hide` é que podemos adicionar efeitos para quando clicarmos no ícone da lixeira. Para fazermos com que a eliminação da tarefa seja feita em câmera lenta, vamos passar o parâmetro `'slow'` para o método `hide`.

```
function onTarefaDeleteClick() {
  $(this).parent('.tarefa-item')
    .hide('slow');
}
```

Agora vamos clicar no ícone da lixeira e apreciar a animação do item sendo eliminado da nossa lista.

O método `hide` permite que um segundo argumento seja passado. Ele é uma função que será executada assim que a animação terminar, um callback. Vamos aproveitar isso e passar uma função que elimina de vez o item da tarefa de dentro do DOM:

```
function onTarefaDeleteClick() {
  $(this).parent('.tarefa-item')
    .hide('slow', function() {
      $(this).remove();
    });
}
```

Mas, uma vez que eu não vejo mais a tarefa na tela, preciso mesmo apagá-lo do DOM? Eu insisto tanto nessa questão de eliminarmos o elemento do DOM pelo simples fato de que, com o tempo, o nosso DOM vai ficar abarrotado de elementos que não servem para mais nada, ocupando cada vez mais memória e deixando nosso código mais lento. Eliminando o que nunca mais vai ser usado é uma forma de mantermos a casa limpa e o consumo de memória sempre baixo.

Muita atenção ao usar o this

No nosso código, perceba que usamos o `this` duas vezes, mas em cada uma delas estávamos falando de elementos diferentes. O `this` é uma palavra reservada que indica qual elementos estamos manipulando no momento.

No primeiro `this` estamos tratando um evento do ícone da lixeira, então ele está se referindo ao elemento `div` que tem a classe `tarefa-delete`. Já no segundo item estamos tratando um evento do item da lista de tarefas, então estamos nos referindo ao elemento que contém a classe `tarefa-item`.

O modo como o `this` funciona é uma das partes confusas do JavaScript mas, prestando atenção ao código e entendendo como funciona essa palavra chave, podemos evitar muita dor de cabeça no futuro.

6.5 Editando itens

Para editar uma tarefa queremos que seja suficiente clicar em cima do item. Ao clicar, devemos fazer com que apareça uma caixa de texto onde o usuário pode alterar o conteúdo, pressionando *ENTER* em seguida para gravar o novo texto.

Vamos começar declarando o evento dentro do método `$(document).ready` e, seguindo o padrão que explicamos, criaremos uma função chamada `onTarefaItemClick` para ser executada quando o evento `click` for disparado por algum elemento da classe `tarefa-item`:

```
$(function() {
  function onTarefaItemClick() {
    console.log("Aqui vamos editar a tarefa");
  }
```

```
  $(".tarefa-item").click(onTarefaItemClick);
});
```

Vamos guardar o texto da tarefa numa variável para, em seguida, criarmos um campo de texto chamado `tarefa-edit` para editá-lo. Note como alteramos o conteúdo da nossa tarefa substituindo o código HTML que fica dentro do elemento. Para isso nós montamos o HTML dentro de uma variável e substituímos o conteúdo da nossa tarefa usando o método `html`.

```
function onTarefaItemClick() {
  var text = $(this).children('.tarefa-texto').text();

  var html = "<input type='text' " +
             "class='tarefa-edit' value='" +
             text + "'>";

  $(this).html(html);

  $(".tarefa-edit").keydown(onTarefaEditKeydown);
}
```

Precisamos também criar uma função, que vamos chamar de `onTarefaEditKeydown` para salvar a alteração da tarefa quando o usuário pressionar *ENTER*.

```
function onTarefaEditKeydown(event) {
  if(event.which === 13) {
    savePendingEdition($(this));
  }
}
```

Vamos criar também uma função chamada `savePendingTarefa`, que vai se encarregar de salvar na lista o texto da tarefa que estamos editando.

```
function savePendingEdition(tarefa) {
  console.log("E aqui vamos salvar nossa tarefa");
}
```

Note que precisamos declarar os eventos dos elementos recém-criados. Isso acontece porque aquele nosso código que é executado no `jQuery(document).ready` é disparado apenas uma vez, quando o browser termina de carregar o documento. Após isso, toda alteração que precisarmos fazer no DOM deve ser seguida da respectiva declaração de eventos.

6.6 Editando apenas um item de cada vez

Agora vamos testar clicando na tarefa e veremos uma caixa de texto com ela lá dentro, pronta para ser alterada.

Mas faça o seguinte teste: clique uma vez dentro dessa caixa de texto e você vai ver que o conteúdo da nossa tarefa desaparece. Temos um problema!

Isso acontece porque, ao clicar na tarefa a ser editada, você acaba disparando novamente uma chamada à função `onTarefaItemClick`. Dessa vez ela não vai encontrar nenhum elemento com a classe `tarefa-texto` dentro daquele item, e vai entender que o texto da sua tarefa está vazio. Na sequência será criada numa nova caixa de texto com o texto vazio e você acabou de perder o conteúdo antigo da sua tarefa!

O que fazer?

Primeiro, precisamos saber se já estamos editando ou não uma tarefa. Vamos criar uma variável `$lastClicked` para armazenar a tarefa que estamos editando. Se a tarefa que clicarmos for a mesma que já estamos alterando, vamos ignorar o clique. Se clicarmos em outra tarefa, a que está sendo editada deve ser salva e a outra deve ser aberta para alteração.

Dessa forma teremos certeza de que somente uma tarefa será editada por vez, e de que não vamos perder o que estamos escrevendo por causa de um erro simples.

```
$(function() {
  var $lastClicked;

  // aqui continua todo o código que já escrevemos

});
```

Dentro da função `onTarefaItemClick`, vamos verificar se o item a ser editado agora é o mesmo que já estamos editando, ou se é algum item novo. O jQuery nos oferece o método `is` para verificarmos se o elemento é o mesmo. Caso seja, simplesmente vamos ignorar o evento e agir como se nada tivesse acontecido.

Temos que verificar também se a variável `$lastClicked` é `undefined`. Nesse caso, não estamos alterando nenhuma tarefa ao entrar nessa função e não precisamos nos preocupar em salvar nada.

```
function onTarefaItemClick() {
  if(!$(this).is($lastClicked)) {
```

```
    if($lastClicked !== undefined) {
      savePendingEdition($lastClicked);
    }

    $lastClicked = $(this);
```

No final da função, vamos usar o mesmo código que já tínhamos antes, mas alterando o `$(this)` para `$lastClicked` que, a essa altura do código, significam a mesma coisa.

```
// continuando onTarefaItemClick

    var text = $lastClicked.children('.tarefa-texto').text();
    var html = "<input type='text' class='tarefa-edit' value='" +
      text + "'>";

    $lastClicked.html(content);

    $(".tarefa-edit").keydown(onTarefaEditKeydown);
  }
}
```

Agora que criamos essa variável `$lastClicked`, precisamos alterar também aquela função `onTarefaEditKeydown`, para continuarmos gravando o texto da tarefa correta.

```
function onTarefaEditKeydown(event) {
  if(event.which === 13) {
    savePendingEdition($lastClicked);
    $lastClicked = undefined;
  }
}
```

Agora sim ficou bonito. Mas lembra do evento que adicionamos para excluir uma tarefa?

6.7 Disparando mais de um evento ao mesmo tempo

Como o ícone da lixeira fica **dentro** do nosso item da lista de tarefas, ao clicarmos ali dispararemos dois eventos, o `click` do elemento que contém a classe

`tarefa-delete` e também o `click` do elemento `tarefa-item`, que acabamos de desenvolver.

O problema é que, enquanto executamos aquele efeito bacana ao excluir uma tarefa, nós também vamos ver uma caixa de texto sendo criada, como se estivéssemos editando a tarefa. Na sequência a tarefa desaparece e, com ela, a caixa de texto. Nada que atrapalhe a nossa vida, mas convenhamos que o efeito fica bem feio.

Não há segredo para solucionar isso. Primeiro, precisamos desassociar todos os eventos `click` do elemento com a classe `tarefa-item`. Como já vimos, nós podemos resolver isso usando o método `off`. Usando também a boa prática do encadeamento de métodos, nosso código vai ficar com essa cara:

```
function onTarefaDeleteClick() {
  $(this).parent('.tarefa-item')
    .off('click')
    .hide('slow', function() {
      $(this).remove();
    });
}
```

6.8 Salvando a tarefa

Ficou faltando escrevermos a função que salva a tarefa que está sendo editada. Nos códigos acima, simplesmente mandamos exibir uma mensagem no console.

Salvar a tarefa significa que vamos pegar o que foi escrito na caixa de texto e montar um item da lista de tarefas, com a possibilidade de permitir a edição quando o usuário clicar no texto, e também que seja excluído quando o usuário clicar no ícone da lixeira.

Vamos começar guardando o texto em uma variável chamada `text`. Em seguida vamos eliminar todo o código HTML que está dentro do elemento `div` que contém o item da tarefa usando o método `empty`.

```
function savePendingEdition($tarefa) {
  var text = $tarefa.children('.tarefa-edit').val();
  $tarefa.empty();
```

Vamos adicionar aqueles três elementos `div` que foram explicados lá no início do capítulo: um que vai conter o texto da tarefa, um que vai exibir o ícone da lixeira e um que serve apenas para limpar a formatação do CSS e deixar os componentes alinhados na tela.

Para adicionar os elementos, vamos usar o método `append`, que adiciona conteúdo ao HTML do elemento.

Por exemplo, o código `$("<b></b>").append("bacana")` vai retornar o HTML `<b>bacana</b>`.

```
//
  $tarefa.append("<div class='tarefa-texto'>" + text + "</div>")
      .append("<div class='tarefa-delete'></div>")
      .append("<div class='clear'></div>");
```

E, finalmente, vamos declarar os eventos para o elementos recém criados, fazendo com que a tarefa seja editada ao ser clicada, ou excluída quando o usuário clicar no ícone da lixeira.

```
//
  $(".tarefa-delete").click(onTarefaDeleteClick);

  $tarefa.click(onTarefaItemClick);
}
```

Agora a nossa lista de tarefas permite edição de um item por vez e exclusão com um efeito batuta. Certamente precisaríamos depois enviar essas informações para o servidor, por exemplo através de AJAX. Veremos como trabalhar com o servidor em breve, usando as facilidades do jQuery.

Precisamos finalizar o nosso desenvolvimento permitindo a inclusão de tarefas.

6.9 Indo ainda mais fundo na manipulação de DOM

Vamos aproveitar o código que utilizamos para adicionar os elementos `div` no nosso item de tarefas para ensinar mais maneiras de manipular o DOM usando jQuery.

Listagem 6.4 - O nosso código original:

```
$tarefa.append("<div class='tarefa-texto'>" + text + "</div>")
    .append("<div class='tarefa-delete'></div>")
    .append("<div class='clear'></div>");
```

Não estamos dizendo aqui que é uma boa prática, que é mais rápido ou que seja mais simples de ler, apesar por eu achar que é mais legível, mas é interessante que

você conheça na prática mais formas de utilizar o jQuery. E, no final das contas, quanto mais pretextos para que você aprenda, melhor.

O jQuery permite que você manipule um elemento do DOM simplesmente passando o código HTML como parâmetro para a função $(). Vamos usar um elemento div para testar. Note que tanto faz escrevermos <div /> ou <div></div>. Vamos executar o exemplo a seguir no console.

Vamos usar o método addClass para adicionar uma classe *CSS* ao nosso elemento HTML e o já conhecido método text para adicionarmos um texto nesse mesmo elemento.

```
$("<div />")
  .addClass("tarefa-texto")
  .text("Lavar o banheiro");
```

Podemos ver o código HTML que foi gerado na figura 6.8.

```
> $("<div />")
      .addClass("tarefa-text")
      .text("Lavar o banheiro");
  [<div class="tarefa-text">Lavar o banheiro</div>]
> |
```

Figura 6.8: Lavar o banheiro. =(

Extrapolando esse exemplo de volta para o nosso código da função savePendingEdition, podemos reescrevê-lo da seguinte forma, adicionando o que aprendemos aqui ao objeto $tarefas, que recebemos por parâmetro:

```
$tarefa.append($("<div />")
                 .addClass("tarefa-texto")
                 .text("Lavar o banheiro"))
       .append($("<div />")
                 .addClass("tarefa-delete"))
       .append($("<div />")
                 .addClass("clear"));
```

6.10 Adicionando tarefas

Para adicionar uma tarefa, o usuário digita algo numa caixa de texto que tem o ID tarefa e pressiona *ENTER*. Em seguida adicionamos na lista uma tarefa com o texto que foi digitado e a caixa de texto volta a ficar em branco.

Usando o que já aprendemos sobre como capturar a tecla *ENTER*, vamos ter um código como esse:

```
function onTarefaKeydown(event) {
  if(event.which === 13) {
    addTarefa($("#tarefa").val());
    $("#tarefa").val("");
  }
}
```

Note que jogamos o trabalho pesado para uma função `addTarefa`. Nessa função nós vamos criar um elemento `div` com a classe `tarefa-item`, e em seguida adicionar aqueles três elementos `div` que já apresentamos: um que contém o texto, outro para o ícone da lixeira e um terceiro que existe apenas para fins de formatação.

Por uma questão de legibilidade, e para usar o que já aprendemos, vamos escrever o código HTML usando o conteúdo da seção anterior. Ao terminarmos de gerar os elementos da nossa tarefa, vamos adicioná-lo ao elemento que representa a lista de tarefas, que vai ter a classe `tarefa-lista`.

Finalmente, vamos adicionar os eventos para quando o usuário clicar no ícone da lixeira e para quando o usuário clicar na tarefa para editá-la.

```
function addTarefa(text) {
  var $tarefa = $("<div />")
                .addClass("tarefa-item")
                .append($("<div />")
                        .addClass("tarefa-texto")
                        .text(text))
                .append($("<div />")
                        .addClass("tarefa-delete"))
                .append($("<div />")
                        .addClass("clear"));

  $("#tarefa-list").append($tarefa);

  $(".tarefa-delete").click(onTarefaDeleteClick);

  $(".tarefa-item").click(onTarefaItemClick);
}
```

Caso você tenha achado confuso, você pode usar a forma tradicional, que é criar uma variável contendo todo o HTML e depois inserir de uma vez só usando o mé-

todo append. Eu acho essa segunda forma mais bagunçada, mas fique à vontade para usar a forma que você achar mais expressiva e fácil de entender.

Com a forma tradicional ficaria assim:

```
function addTarefa(text) {
  var tarefa = '<div class="tarefa-item">'
    + '<div class="tarefa-texto">' + text + '</div>'
    + '<div class="tarefa-delete"></div>'
    + '<div class="clear"></div>'
    + '</div>';

  $("#tarefa-list").append(tarefa);

  $(".tarefa-delete").click(onTarefaDeleteClick);

  $(".tarefa-item").click(onTarefaItemClick);
}
```

Note que, na segunda forma, eu chamei a variável de tarefa ao invés de $tarefa, pois estamos lidando com String, e não com um objeto jQuery.

Agora, depois de todo o nosso esforço, temos uma lista de tarefas totalmente funcional. Uma demonstração do código apresentado está no endereço abaixo:

http://jsfiddle.net/pbalduino/faWn4/

No próximo capítulo vamos aprender a nos comunicar com o servidor sem que seja necessário recarregar a página. Assim poderemos não só enviar informações sobre o que estamos fazendo, como também receber dados do servidor e atualizar a nossa página de modo rápido e agradável ao usuário.

Capítulo 7

Não tenha medo do AJAX e do JSON

"AJAX não é uma tecnologia, mas sim um conjunto formado por várias tecnologias, cada uma se desenvolvendo por conta própria, se juntando de formas novas e poderosas."
– Jesse James Garrett

Por mais que já tenhamos trabalhado com AJAX e JSON, sempre há pontos a conhecer melhor.

7.1 AJAX? Que bicho é esse?

Antigamente, cada vez que você quisesse atualizar alguma informação na sua página, por menor que fosse, você precisava atualizar toda a página. Isto é, cada clique fazia o navegador carregar uma nova página inteiramente, carregando cabeçalhos, rodapés, conteúdo em comum e às vezes até mesmo imagens, javascripts e css. Um trabalho

desgastante tanto para o cliente quanto para o servidor. Mais ainda: fica notável, para o usuário, uma certa lentidão. O site acaba "respondendo" sem muita velocidade.

Tudo começou a mudar com, quem diria, o lançamento do Internet Explorer 5, que implementava um objeto ActiveX chamado **XMLHTTP**. Apesar de não ter chamado a atenção na época, esse objeto foi logo implementado no Mozilla (o antecessor do Firefox), Opera e outros browsers através do objeto JavaScript `XMLHttpRequest`. Por sua vez, ele apareceu dessa mesma forma no Internet Explorer 7 [18].

Esse tal objeto `XMLHttpRequest` permite que seu documento HTML acesse outras páginas ou arquivos por baixo dos panos, trazendo informações atualizadas sem a necessidade de recarregar toda uma página.

Somando isso à manipulação de DOM, que já vimos nos capítulos anteriores, que permite que porções do documento sejam atualizados sem a necessidade de se recarregar a página, foi criado o que conhecemos por `AJAX`. Utilizando AJAX, podemos fazer o nosso site trabalhar da maneira que planejamos no capítulo 2, quando apresentamos o gráfico 2.2 de requisições ao servidor e atualizações no navegador.

Esse termo foi cunhado pelo especialista em experiência de usuário Jesse James Garrett, em 2005, em seu artigo *Ajax: A New Approach to Web Applications*[7].

Nesse artigo, Jesse James (que, segundo o próprio, não tem qualquer relação com o ladrão de bancos americano[8]) explica que AJAX não é uma, mas um conjunto de tecnologias que incluem a manipulação do DOM e o acesso a dados de um modo transparente para o usuário.

A título de curiosidade, o termo `AJAX` é um acrônimo para **Asynchronous JavaScript and XML**, apesar de atualmente o XML ser pouco utilizado e de que não há obrigatoriedade de que o processo seja assíncrono.

Síncrono e assíncrono

Imagine que você lidera uma equipe e pede para João, um de seus programadores, que corrija um bug no sistema, avisando assim que terminar a tarefa.

Enquanto ele trabalha, você está livre para participar de uma reunião, responder emails, telefonar para alguns fornecedores, sair para tomar um café e dar bronca em algum outro membro da equipe.

Enquanto isso, você recebe um email enviado por João informando que o bug está corrigido. Perceba que você disparou uma tarefa e ficou livre para fazer várias outras coisas enquanto ela não estava pronta.

Como não foi necessário ficar parado enquanto a tarefa era executada, dizemos que ela é **assíncrona**.

Imagine agora que você passe outra tarefa para o João e tenha que ficar sentado ao lado dele, esperando que ela seja concluída. Enquanto isso você não pode fazer mais nada até que João termine o que foi passado. Dizemos então que essa tarefa é **síncrona**.

Normalmente o uso de tarefas assíncronas faz com que a sua aplicação termine o conjunto de tarefas mais rapidamente.

Por outro lado, como líder da equipe, você percebeu que nem sempre é bom deixar João trabalhando sozinho e sem supervisão.

Isso tem algo a ver com a tal Web 2.0?

Tem sim. O termo Web 2.0 estava ligado ao uso de diversas práticas na Web, e o AJAX era uma dessas práticas. O termo Web 2.0 já perdeu a força. Até mesmo a sigla AJAX, por ser basicamente onipresente nas aplicações web modernas, faz menos barulho. Além disso, é bastante raro, senão impossível, ver alguém manipulando diretamente o objeto `XMLHttpRequest`, dada sua complexidade e incompatibilidade entre diferentes navegadores.

Para executar nossos códigos JavaScript sem precisar instalar ou configurar um servidor, vamos usar um site chamado jsFiddle, que vamos explicar a seguir.

7.2 Usando o jsFiddle

O `jsFiddle` é um site que nos permite escrever código JavaScript, HTML e CSS na mesma tela, executar e ver o resultado imediatamente, sem que haja necessidade de instalarmos nada na máquina.

Para executar os exemplos a seguir, selecione a última versão disponível do jQuery onde está escrito *Choose Framework*, conforme a figura 7.1.

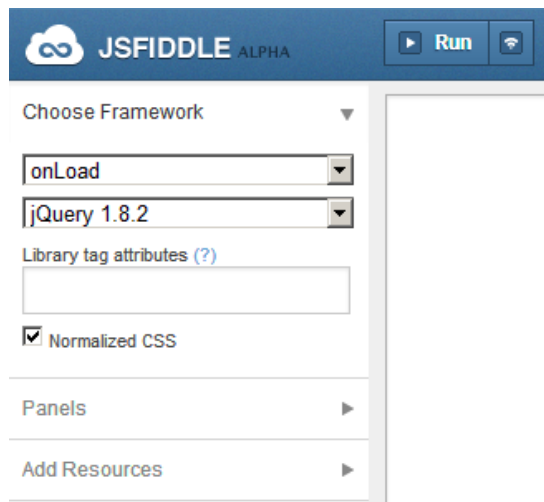



Figura 7.1: Selecione o framework

A interface do jsFiddle, conforme podemos ver na figura 7.2 é composta por quatro painéis: uma para você digitar o código `HTML` e outra para o código `CSS` na parte de cima, uma para o seu código `JavaScript` e outra para o resultado na parte de baixo. Os quadros vêm identificados e você não vai ter problema nenhum em se localizar. No topo da tela existe um botão **Run**, que serve para executar os códigos que você inseriu e exibir o resultado.

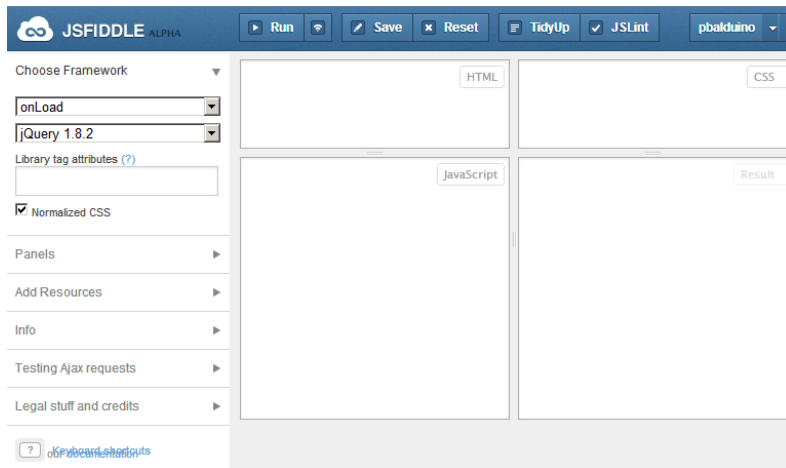



Figura 7.2: Interface do jsFiddle

Vamos escrever um código bem simples para entendermos melhor como usar a ferramenta.

No painel em que estiver escrito JavaScript, que costuma ser o primeiro da linha de baixo, escreva o código abaixo. Em seguida, clique em **Run**, que é o primeiro botão no topo da página.

```
alert("Olá jsFiddle!");
```

Ao executar veremos a mensagem, como na figura 7.3

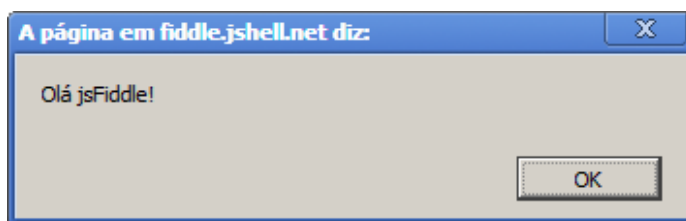



Figura 7.3: jsFiddle em funcionamento

Experimente brincar um pouco com essa ferramenta. Eu a considero muito útil para demonstrar recursos do JavaScript ou mesmo reproduzir erros de alguma biblioteca que você esteja usando.

7.3 Nosso primeiro código com AJAX

Como já dissemos, apesar do **X** do **AJAX** significar **XML**, o retorno do servidor pode ser qualquer informação, mesmo em forma texto puro.

Vamos aprender a executar a forma mais simples de requisição AJAX usando jQuery. Primeiro, vamos acessar o endereço abaixo:

http://livro-capitulo07.herokuapp.com/hello

E vamos ver a saída da figura 7.4. Podemos ver que a resposta é um texto simples, sem formatação nem nada de especial.

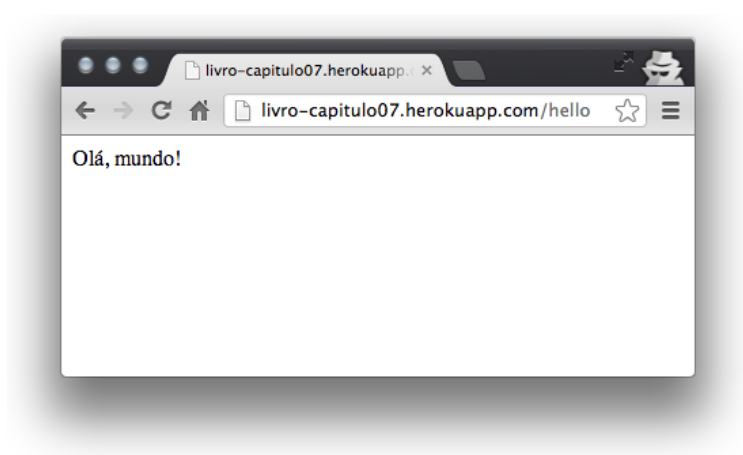



Figura 7.4: Olá!

Vamos fazer com que nosso código carregue esse texto usando AJAX e o exiba na nossa tela.

O jQuery nos disponibiliza um método `get`, que aceita até três parâmetros.

O primeiro contém a `URL`, o endereço do site que vamos acessar para pegar os dados. A página que nos fornece as informações para serem usadas com AJAX é chamada **serviço**.

O segundo parâmetro, opcional, é onde você informa os parâmetros a serem passados para o serviço. Como estamos usando o *AJAX mais simples que poderia funcionar*, vamos deixar para explicar esses parâmetros mais para frente.

No terceiro parâmetro, também opcional, nós passaremos um callback que recebe como parâmetro os dados que foram lidos do serviço. Mais para frente vamos

explicar também como trabalhar sem esse terceiro parâmetro, mas no momento vamos manter as coisas simples.

Vamos então acessar o serviço que está em http://livro-capitulo07.herokuapp.com/hello e exibir o texto que é retornado:

```
var servico = "http://livro-capitulo07.herokuapp.com/hello";

$.get(servico, function(data) {
  alert(data);
});
```

Ao executarmos esse código, veremos a mensagem da imagem 7.5

Figura 7.5: Olá!

7.4 Enviando parâmetros com AJAX

Lembra que eu falei sobre o segundo parâmetro, que é opcional, onde podemos passar parâmetros para o serviço? Vamos aprender a usá-lo agora para retornar uma mensagem um pouco diferente.

O endereço do serviço é o mesmo, mas se passarmos um nome para ele, usando um parâmetro chamado `nome`, a mensagem de retorno será diferente, repetindo o nome que você informou.

O nosso parâmetro vai ser definido como um *dicionário*, separando o nome do valor utilizando o caractere de dois pontos.

Dicionário

Dicionários, como você provavelmente já sabe, são estruturas comuns em várias linguagens e funcionam como *Arrays*, com a diferença que, em um dicionário, o identificador pode ser qualquer valor, enquanto em um Array o identificador é sempre um número. É o hashtable, ou mapa, de algumas outras linguagens. Neste livro podemos utilizar os termos *objeto* e *dicionário* como sinônimos, a menos que eu esteja explicitamente dizendo o contrário.

Se declararmos uma variável contendo um dicionário como no código abaixo

```
var parametros = {nome: "Paulo", idade: 33};
```

Quando acessarmos o valor usando `parametros["nome"]`, teremos o texto **"Paulo"**. Analogamente, `parametros["idade"]` devolve `33`.

Vamos definir o nome a ser passado da forma a seguir:

```
var parametros = {nome: "Caro Leitor"};
```

Agora passamos o parâmetro no método `get`:

```
var parametros = {nome: "Caro Leitor"};
var servico = "http://livro-capitulo07.herokuapp.com/hello";

$.get(servico, parametros, function(data) {
  alert(data);
});
```

Agora, veja só, a figura 7.6 mostra que a mensagem foi alterada de acordo com o parâmetro que passamos.

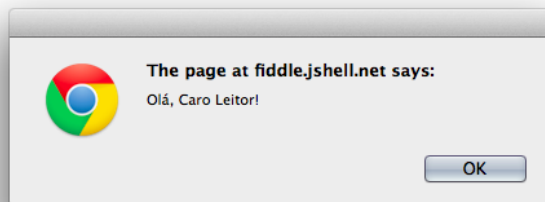



Figura 7.6: Mensagem alterada

Note como é simples fazer requisições AJAX para um servidor usando parâmetros quando necessário.

Como foi dito antes, o terceiro parâmetro daquele método `get` também é opcional.

```
$.get(servico, parametros);
```

Ao executarmos esse código, vamos ver que... não aconteceu nada. Se não passarmos um callback para ser executado quando o valor retornar, como vamos saber o que retornou, ou mesmo se o serviço retornou alguma coisa?

Pois é aqui que vamos aprender como usar AJAX de uma forma mais profissional no jQuery.

7.5 O objeto jqXHR

Vamos descobrir como trabalhar com AJAX de um modo mais sofisticado, que permita inclusive que possamos tratar a resposta do servidor de acordo com ela. Podemos executar um código caso o servidor retorne um erro e outro código se tudo correr bem.

No início do capítulo falamos sobre os objetos `XMLHTTP`, usado pelo Internet Explorer até a versão 7, e `XMLHttpRequest`, usados pelos browsers modernos e pelo Internet Explorer a partir da versão 8. Para evitar que o desenvolvedor tenha que ficar lidando com um ou com outro, o jQuery nos oferece um objeto `jqXHR`, que é o modo curto que os desenvolvedores do jQuery acharam para nomear o **jQuery XMLHttpRequest Object**.

Apesar da sopa de letrinhas, decorar o nome não vai ser importante para nós.

O importante é você saber que esse objeto pode intermediar e controlar toda a conversação entre o seu código JavaScript e o servidor que vai nos fornecer os dados. Nós o recebemos como retorno toda vez que usamos algum método do jQuery que efetua uma requisição AJAX, como por exemplo, o método `get`, que vimos nos exemplos anteriores.

O objeto `jqXHR` permite que sejam definidos callbacks para quando a requisição correu bem e para quando houve algum erro. Vamos reescrever os exemplos anteriores para entender como isso funciona.

Para tratar os dados recebidos quando tudo corre bem, vamos usar o evento `done`.

```
var parametros = {nome: "Caro Leitor"};
var servico = "http://livro-capitulo07.herokuapp.com/hello";

var $xhr = $.get(servico, parametros);

$xhr.done(function(data) {
  alert(data);
});
```

Ao executarmos, veremos a mesma resposta da figura 7.5. Só que esse código pode ficar ainda melhor. Essa nova variável `xhr` que criamos pode ser evitada, concatenando chamadas de funções:

```
var parametros = {nome: "Caro Leitor"};
var servico = "http://livro-capitulo07.herokuapp.com/hello";

$.get(servico, parametros)
    .done(function(data) {
      alert(data);
    });
```

É importante lembrar que devemos tomar cuidado em encadear muitas funções dessa forma, evitando tornar o código ilegível.

Figura 7.7: Erro 404 é um dos mais comuns

Podemos criar um callback específico para quando o servidor nos retornar um erro. Eu criei um endereço que vai nos retornar erro toda vez que o acessarmos. Poderíamos simplesmente informar um endereço que não existe, mas preferi não complicar as coisas. O evento que trata os casos de erro é o `fail`.

```
var servico = "http://livro-capitulo07.herokuapp.com/error";

$.get(servico)
    .fail(function(data) {
      alert(data.responseText);
    });
```

Vamos ver então a mensagem da figura 7.8.

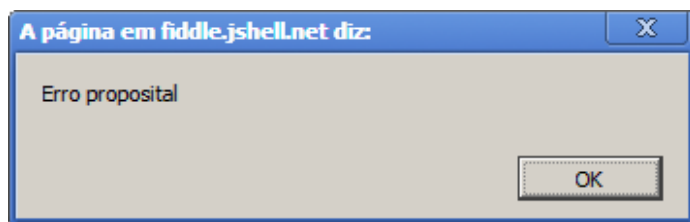



Figura 7.8: Foi mesmo um erro proposital

Mas o que é considerado um erro?

Toda requisição `HTTP`, que é o protocolo sobre o qual o AJAX trabalha, retorna um código de três dígitos que nos informa se tudo correu bem, se a página não foi encontrada, se o servidor apresentou erro ou mesmo se você deve ser redirecionado para outro endereço.

Quando a requisição traz um código que começa com 1, 2 ou 3, dizemos que a comunicação ocorreu com sucesso. O código **200**, por exemplo, significa que tudo correu bem, o conteúdo foi enviado, e normalmente vem acompanhado do texto "OK".

Quando retorna com um código que começa com 4 ou 5, sabemos que houve algum erro na requisição. Os erros mais comuns são o **404**, que indica que o recurso ou a página não foi encontrada, e o erro **500**, que indica que houve um erro qualquer no servidor.

Uma lista de códigos HTTP pode ser encontrada em http://goo.gl/VJz74

Já sabemos como transportar texto de um lado para o outro. Também ficou fácil descobrir se a chamada AJAX foi feita com sucesso. Agora falta saber como organizar os dados que vamos transmitir de um lado para o outro. Faremos isso com JSON.

7.6 E o que é o JSON?

"Eu não digo que inventei o JSON... O que eu fiz foi encontrar, nomear, descrever como aquilo era util... A ideia já estava entre nós há algum tempo. O que eu fiz foi criar a especificação e criar um pequeno site sobre o assunto."
– Douglas Crockford

Na maioria das vezes, transmitir apenas uma frase ou um número não é o suficiente para que nosso código JavaScript se comunique com o servidor. Nesses casos, precisamos enviar e receber listas, valores e objetos complexos contendo toda a informação necessária. Existem inúmeras soluções para isso, sendo que as mais comuns são XML e JSON.

O formato XML não é simples de ser lido por humanos, e as coisas só pioram se estivermos lidando com objetos contendo objetos. Já `JSON` é uma forma de serializar, ou converter para texto, objetos JavaScript de modo que sejam legíveis inclusive por humanos.

Esse formato foi especificado e definido originalmente por Douglas Crockford, aquele famoso autor do livro *JavaScript: The Good Parts*[4], por volta do ano de 2001.

A grande vantagem do JSON sobre qualquer outra forma de serialização é que o JavaScript lida com o formato de modo transparente, convertendo a informação em formato texto para um tipo nativo da linguagem, evitando o uso de bibliotecas e ferramentas que possam complicar o seu código.

Podemos trabalhar com seis tipos de dados: **textos**; **números**, que podem ser inteiros ou decimais; **lógicos**, ou booleanos, que são `true` ou `false`; **listas**, que funcionam como `Arrays` JavaScript; **objetos**, que funcionam como `dicionários` JavaScript, conforme vimos neste mesmo capítulo; e `null`, que simboliza a ausência de um valor.

Você pode combinar livremente objetos e listas com os seis tipos de dados válidos, representando valores complexos que contêm toda a informação necessária para a sua requisição.

Mais detalhes e a especificação completa podem ser encontrados em http://json.org/

Para demonstrar que cara tem o JSON, vamos usar os dados desse livro para que você entenda melhor como funciona:

```
{
  "autor"     : "Plínio Balduino",
  "titulo"    : "Dominando JavaScript com jQuery",
  "ano"       : 2013,
  "editora"   : "Casa do Código",
  "usa_drm"   : false,
  "capitulos": [
                "Apresentação",
                "Refazendo uma loja virtual",
```

```
                    "Adicionando JavaScript",
                    "Um JavaScript diferente em cada navegador",
                    "Simplifique com jQuery"
                  ]
}
```

Um recurso muito bacana que pode nos ajudar a visualizar melhor um objeto JSON é o site http://www.jsoneditoronline.org/

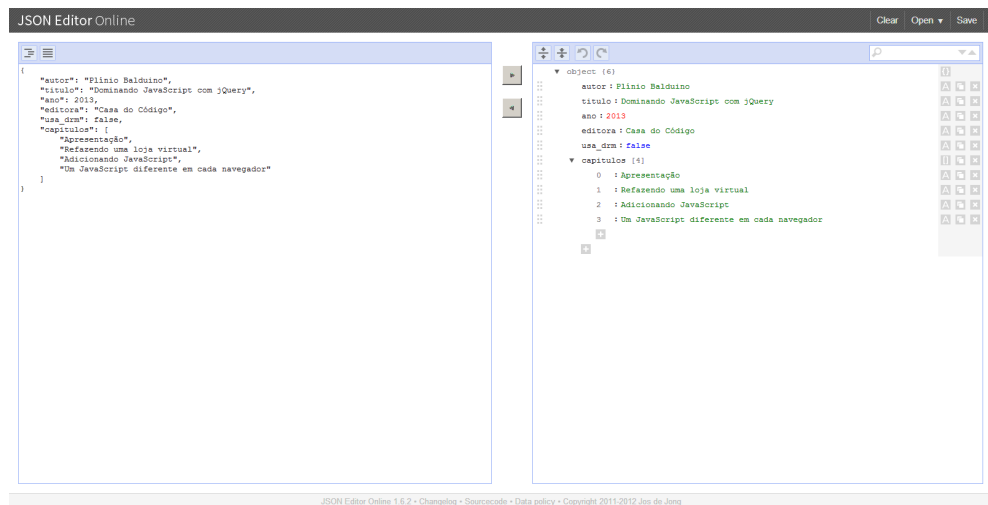



Figura 7.9: JSON editor online

Esse site tem dois painéis. No da esquerda você escreve o código JSON, pressiona o botão com a seta para a direita e, no painel da esquerda, você vai ver uma representação interativa do objeto.

É possível também montar um objeto do lado direito, pressionar o botão com a seta para o lado esquerdo e ver o código JSON equivalente no painel esquerdo. Ao entrarmos com o código JSON do nosso livro, conforme demonstramos acima, teremos a representação visual da figura 7.10

```
▼ object {6}
    autor : Plínio Balduino
    titulo : Dominando JavaScript com jQuery
    ano : 2013
    editora : Casa do Código
    usa_drm : false
  ▼ capitulos [4]
      0 : Apresentação
      1 : Refazendo uma loja virtual
      2 : Adicionando JavaScript
      3 : Um JavaScript diferente em cada navegador
```

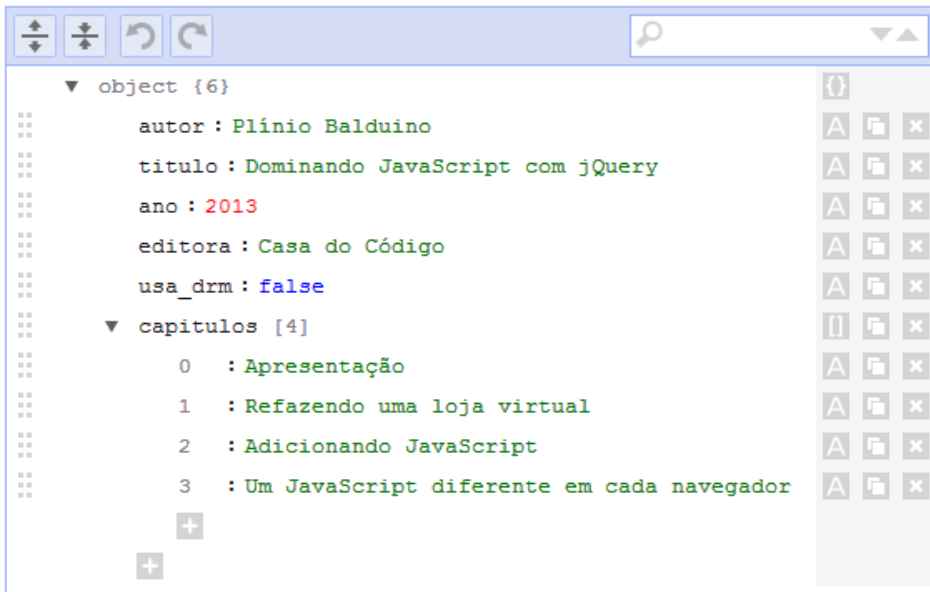



Figura 7.10: Representação do código JSON

Lembra que anteriormente nós falamos sobre *dicionários* em JavaScript? Se olharmos com mais atenção ao conteúdo do nosso código JSON, vamos ver que seu conteúdo nada mais é do que um *dicionário*, contendo *Strings*, valores numéricos e lógicos, ou booleanos. Esse *dicionário* contém também um *Array* com alguns capitulos do livro.

Não por coincidência, o formato JSON é um código JavaScript válido, podendo ser atribuído a uma variável e tratado como um objeto qualquer, como fizemos na figura 7.11. Dessa forma, podemos acessar cada valor usando a chave como uma propriedade de um objeto JavaScript. Para acessar os nomes dos capítulos do livro, podemos acessar cada um deles como um `Array` do JavaScript, exatamente porque **JSON é JavaScript puro**.

```
> var livro = {
    "autor"    : "Plínio Balduino",
    "titulo"   : "Dominando JavaScript com jQuery",
    "ano"      : 2013,
    "editora"  : "Casa do Código",
    "usa_drm"  : false,
    "capitulos": [
                   "Apresentação",
                   "Refazendo uma loja virtual",
                   "Adicionando JavaScript",
                   "Um JavaScript diferente em cada navegador",
                   "Simplifique com jQuery"
                 ]
  };
  undefined
> livro.titulo;
  "Dominando JavaScript com jQuery"
> livro.capitulos[2];
  "Adicionando JavaScript"
```

Figura 7.11: Usando JSON como um objeto JavaScript

7.7 Juntando JSON e AJAX

Usar AJAX com JSON é algo muito simples com jQuery.

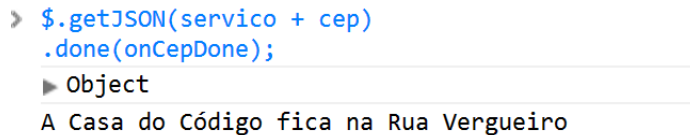
Vamos usar um site de consulta de CEP chamado Postmon para demonstrar. Ao informar um CEP, o serviço vai retornar um JSON com os dados do logradouro se o código for válido, ou um erro se o CEP não for encontrado na base dos Correios.

Para acessarmos o serviço usando o CEP *04101-300*, vamos usar o endereço abaixo:

http://api.postmon.com.br/v1/cep/04101-300

O serviço vai retornar o conteúdo abaixo, em formato JSON. Formatei o resultado para ficar mais fácil entendermos.

```
{
  "complemento": "de 2771 a 5049 - lado \u00edmpar",
  "bairro"     : "Vila Mariana",
  "cidade"     : "S\u00e3o Paulo",
  "logradouro" : "Rua Vergueiro",
  "cep"        : "04101300",
  "estado"     : "SP"
}
```

A única diferença em relação ao que já vimos até agora é que vamos usar o método `getJSON` para retornar um valor JSON. Os parâmetros e o funcionamento são iguais aos do método `get`.

Vamos criar uma função `onCepDone` para exibir o logradouro (nome da rua, praça ou avenida) caso o valor passado seja válido. Aí dizemos que quando a chamada for concluída, o callback `onCepDone` deve ser executado:

```
var servico = "http://api.postmon.com.br/v1/cep/";
var cep = "04101-300";

function onCepDone(data) {
  console.log("A Casa do Código fica na " + data.logradouro);
}

$.getJSON(servico + cep)
  .done(onCepDone);
```

Ao executarmos esse código, teremos parte do endereço da editora, conforme a figura 7.12.

```
> $.getJSON(servico + cep)
  .done(onCepDone);
  ▶ Object
  A Casa do Código fica na Rua Vergueiro
```

Figura 7.12: Retorno da consulta de CEP

Perceba que a variável `data`, que contém o retorno do serviço, já está convertido em um objeto JavaScript. Com isso você pode acessar qualquer informação retornada, como bairro ou cidade, como se uma propriedade do objeto. Foi exatamente o que fizemos para exibir o logradouro.

Vamos preparar nosso código também para lidar com CEPs que não existem na base dos Correios. Para isso, como já vimos antes, vamos usar o evento `fail`.

Aproveitando o código do exemplo anterior, teremos:

```
var servico = "http://api.postmon.com.br/v1/cep/";
var cep = "12345-789";

function onCepDone(data) {
  console.log("A Casa do Código fica na " + data.logradouro);
}
```

```
function onCepError(error) {
  console.log("Erro: " + error.statusText)
};

$.getJSON(servico + cep)
  .done(onCepDone)
  .fail(onCepError);
```

Executando esse código, veremos a mensagem de erro do browser e o conteúdo do erro, que estará na variável `error`, conforme a figura 7.13

```
> $.getJSON(servico + cep)
    .done(onCepDone)
    .fail(onCepError);
  ▶ Object
✖ ▶ GET http://api.postmon.com.br/v1/cep/12345-789 404 (Not Found)
  Erro: Not Found
```

Figura 7.13: Erro no retorno

Para você ver o AJAX funcionando como realmente acontece no dia a dia, basta você criar um `input text` e fazer com que, assim que ele perca o focus, um callback execute o `$.getJSON` passando o argumento da caixa de texto! Se o retorno informar que houve sucesso, pode deixar a borda da caixa de texto verde. Se a requisição retornar algum erro, pode deixar em vermelho! Faça esse exercício!

Repare que nesse caso fazemos a chamada assíncrona: o serviço que verifica o CEP só vai ser chamado depois da página ter sido carregada. Mais ainda, vai acontecer somente quando um evento for disparado e cair na sua função de callback.

É exatamente dessa forma que, por exemplo, a busca do Google preenche sugestões para você na barra de pesquisa. Quando um determinado evento acontece (por exemplo, você digitou mais de 3 caracteres), um callback faz uma chamada para o servidor mandar as sugestões que começam com aquelas 3 letras. No caso de sucesso, o Google lista as opções para você no HTML.

7.8 Políticas de segurança dos browsers

Por que usamos um servidor pra gerar os JSONs que queríamos trabalhar? Porque não podemos fazer isso acessando arquivos locais de teste!

Por uma questão de segurança, os browsers modernos não permitem que você faça chamadas AJAX a partir de um arquivo local ou que você acesse um endereço em

um servidor diferente do seu[19]. A figura 7.14 nos mostra uma mensagem de erro exibida quando tentamos fazer uma chamada AJAX através de um arquivo local, ou seja, que está na nossa própria máquina. Arquivos locais são exibidos na barra de endereços do browser com `file://` ao invés de `http://` no início do endereço.

Existem algumas formas de passarmos por cima dessas políticas de segurança.

```
> $.get("arquivo_qualquer.json");

  ▶ Object
✖ XMLHttpRequest cannot load file:///C:/projetos/js-jquery/rogus-music/site-novo-
  jquery/arquivo_qualquer.json. Origin null is not allowed by Access-Control-Allow-Origin.
> |
```

Figura 7.14: Chamada a partir de um arquivo local

A primeira é chamada de **Cross-Origin Resource Sharing**, e é habilitada quando o servidor que está nos fornecendo os dados insere o cabeçalho `Access-Control-Allow-Origin: *` na resposta HTTP. Essa é a forma mais limpa e padronizada de acessarmos outro domínio usando AJAX, mas tem uma série de problemas e limitações. A maior delas, sem dúvida, é quando estamos acessando um servidor que não inclui esse cabeçalho na resposta. Outro problema é que o Internet Explorer 7 não dá suporte a essa funcionalidade. As versões 8 e 9, de acordo com o site *Can I use...*, dão apenas suporte parcial.

http://caniuse.com/#feat=cors

A segunda forma de se evitar as políticas de segurança é usando uma técnica chamada JSONP, que é um complemento ao formato JSON, e que vamos ver com detalhes mais para frente. Essas duas formas exigem alguma colaboração por parte do servidor.

Existe ainda uma terceira forma, que é a menos segura e **deve ser usada apenas para fins de desenvolvimento e testes**, consiste em desabilitar os mecanismos de segurança do browser.

Se você estiver usando o Chrome, inicie seu browser com o parâmetro `--disable-web-security`.

Se você estiver usando o Firefox, entre no endereço `about:config` e adicione a chave `capability.policy.default.XMLHttpRequest.open` com o valor `allAccess`.

Lembre-se de remover o parâmetro ou a chave assim que você terminar seus

testes. Não queremos que ninguém altere seu perfil no Facebook ou publique algo sem autorização no Twitter, não é mesmo?

No mundo real não podemos ficar desabilitando travas de segurança nas máquinas do cliente, então é sempre bom sabermos como instalar um servidor local para podermos executar nossos scripts enquanto estamos desenvolvendo a aplicação.

Alternativamente, você pode usar o **jsFiddle** para testar seus scripts sem a necessidade de instalar um servidor.

Vamos ver agora como utilizar AJAX com JSON e REST em uma aplicação real.

O que é REST? Você vai aprender no próximo capítulo. Fique atento!

Capítulo 8

Um gerenciador de tarefas com AJAX

"Tudo deveria ser feito da forma mais simples possível, mas não mais simples do que isso."
– Albert Einstein

8.1 Melhorando nosso gerenciador

Agora que aprendemos como usar AJAX e o que é JSON, vamos voltar ao nosso gerenciador de tarefas do capítulo 6 para adicionar mais recursos. Aproveitaremos também para aprender como usar alguns recursos avançados.

Não seria bom se conseguíssemos ver as nossas tarefas mesmo depois de termos fechado o browser ou depois de termos recarregado o browser?

Podemos gravar cada uma das tarefas num servidor que eu preparei especialmente para essa ocasião usando AJAX. As tarefas serão enviadas para o servidor

assim que elas forem atualizadas.

Vamos permitir também que as tarefas sejam lidas assim que a lista de tarefas seja carregada, exibindo todos os itens que estão gravados no servidor.

Usaremos o site *jsFiddle* e o console do browser para escrever e testar os nossos exemplos abaixo. Se você já tiver experiência configurando um servidor web na sua máquina, fique à vontade para executar seus exemplos através dele.

O resultado final do capítulo anterior pode ser encontrado no endereço abaixo, e vai ser a base do que vamos demonstrar neste capítulo.

http://jsfiddle.net/pbalduino/faWn4/

8.2 Carregando as tarefas do servidor

Apesar de estarmos trabalhando no mesmo gerenciador de tarefas, o nosso código HTML vai ficar um pouco diferente do código usado no capítulo 6, uma vez que precisaremos apenas da lista de tarefas vazia.

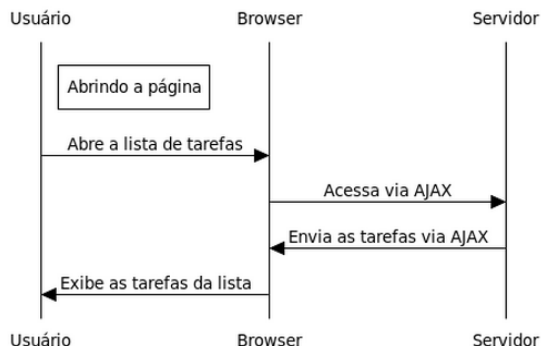



Figura 8.1: Carregando as tarefas

O código desse capítulo está disponível no endereço abaixo:

http://jsfiddle.net/pbalduino/RcRMv/

Vamos apagar tudo o que estiver dentro do elemento `div` que tem o `id` igual a `tarefa-lista`. O `div` com id `tarefa-lista` continua a existir, mas vai ficar vazio.

O nosso código HTML vai ficar assim:

```
<div class="todo">
  <h2>TODO List</h2>
  <input type="text" id="tarefa">
  <div id="tarefa-list">
  </div>
</div>
```

Vamos começar carregando nossas tarefas assim que a lista for exibida pela primeira vez. Para isso, vamos criar e chamar uma função `loadTarefas` dentro do nosso velho amigo, o evento `$(document).ready`.

Como nosso serviço que armazena das tarefas fica num servidor compartilhado, usaremos um identificador para o usuário. Pode ser o seu nome, seu email ou qualquer texto curto que faça as suas tarefas não se misturarem com as dos outros usuários e leitores. Vamos gravar esse identificador numa variável chamada `meu_login`. Vamos gravar também o nome do servidor numa variável chamada (adivinhe?) `servidor`. Isso vai ajudar bastante para o caso de você preferir utilizar o próprio servidor, ou para eventualidade do servidor mudar de endereço.

Com todas essas informações, nosso código vai ficar assim:

```
$(function() {
  var meu_login = "seu@email.com";
  var server = "http://livro-capitulo07.herokuapp.com";

  var $lastClicked;
```

Em seguida temos todo o código do capítulo 6, que vamos relembrar rapidamente para que você não se perca.

Criamos a função `onTarefaDeleteClick`, que será executada sempre que o usuário clicar no ícone da lixeira.

```
function onTarefaDeleteClick() {

  $(this).parent('.tarefa-item')
    .off('click')
    .hide('slow', function() {
      $(this).remove();
    });
}
```

Em seguida temos a função `addTarefa`, que recebe um texto e adiciona a tarefa à lista.

```
function addTarefa(text) {
  var $tarefa = $("<div />")
                .addClass("tarefa-item")
                .append($("<div />")
                        .addClass("tarefa-text")
                        .text(text))
                .append($("<div />")
                        .addClass("tarefa-delete"))
                .append($("<div />")
                        .addClass("clear"));

  $("#tarefa-list").append($tarefa);

  $(".tarefa-delete").click(onTarefaDeleteClick);

  $(".tarefa-item").click(onTarefaItemClick);
}
```

Criamos também dois eventos: um, chamado `onTarefaKeydown`, para quando o usuário pressionar *ENTER* na inclusão de novas tarefas e outro chamado `onTarefaEditKeydown` para quando o usuário pressionar *ENTER* durante a edição da tarefa.

```
function onTarefaKeydown(event) {
  if(event.which === 13) {
    addTarefa($("#tarefa").val());
    $("#tarefa").val("");
  }
}

function onTarefaEditKeydown(event) {
  if(event.which === 13) {
    savePendingEdition($lastClicked);
    $lastClicked = undefined;
  }
}
```

Criamos também um callback chamado `onTarefaItemClick` que será invocado quando o usuário clicar em alguma tarefa, permitindo a alteração.

```
function onTarefaItemClick() {
```

```
  if(!$(this).is($lastClicked)) {
    if($lastClicked !== undefined) {
      savePendingEdition($lastClicked);
    }

    $lastClicked = $(this);

    var text = $lastClicked.children('.tarefa-text').text();

    var content = "<input type='text' class='tarefa-edit' value='" +
      text + "'>";

    $lastClicked.html(content);

    $(".tarefa-edit").keydown(onTarefaEditKeydown);
  }

}
```

Criamos uma função chamada `savePendingEdition` para fazer com que as tarefas que tenham edição pendente sejam gravadas.

```
function savePendingEdition($tarefa) {
  var text = $tarefa.children('.tarefa-edit').val();
  $tarefa.empty();
  $tarefa.append("<div class='tarefa-text'>" + text + "</div>")
        .append("<div class='tarefa-delete'></div>")
        .append("<div class='clear'></div>");

  $(".tarefa-delete").click(onTarefaDeleteClick);

  $tarefa.click(onTarefaItemClick);
}
```

Finalmente vamos atribuir os callbacks aos respectivos eventos.

```
$(".tarefa-delete").click(onTarefaDeleteClick);

$(".tarefa-item").click(onTarefaItemClick);

$("#tarefa").keydown(onTarefaKeydown);
```

E, finalmente, vamos adicionar a função `loadTarefas`, que vai carregar as tarefas que estão gravadas no servidor.

```
function loadTarefas() {
    // calma que já chegaremos aqui
  }

  loadTarefas();
});
```

Na função `loadTarefas`, vamos eliminar todos os itens que por algum motivo estejam na lista e, usando o método `getJSON` vamos informar o endereço do serviço que retornar todas as tarefas daquele usuário. Atenção para as letras maiúsculas ao utilizar os métodos do jQuery, já que o JavaScript diferencia letras maiúsculas de minúsculas.

Para deixar o trabalho o mais simples possível, nosso serviço que vai retornar as tarefas chama-se `tarefas`, e vamos passar um parâmetro `usuario` contendo o nome do usuário que você escolheu no início da explicação.

O método `getJSON` pode receber três parâmetros: o primeiro parâmetro é a URL do serviço ou site que vai fornecer os dados; o segundo contém os parâmetros a serem enviados ao servidor em formato JSON; e o terceiro recebe um callback que será executado assim que o método terminar de receber os dados com sucesso, passando-os por parâmetro para o callback.

Uma forma bem comum e, na minha opinião, mais limpa de se usar esse método é encadear os métodos `done`, para quando a chamada retornar um valor, ou `fail`, para quando acontecer um erro qualquer na comunicação ou na chamada.

O callback do evento `done` vai receber um *Array* contendo todas as tarefas. Você vai receber duas tarefas logo na primeira vez que você executar o exemplo com o seu login, mesmo sem ter cadastrado nenhuma tarefa. O resultado pode ser visto na figura 8.2

```
data:   ▼ [Object, Object]
          ▼ 0: Object
              created_at: "2012-12-10T00:22:09Z"
              id: 2
              texto: "Comprar pão"
              updated_at: "2012-12-10T00:22:09Z"
              usuario: "sample"
            ▶ __proto__: Object
          ▼ 1: Object
              created_at: "2012-12-10T00:22:30Z"
              id: 3
              texto: "Pagar a conta de luz"
              updated_at: "2012-12-10T00:22:30Z"
              usuario: "sample"
            ▶ __proto__: Object
            length: 2
          ▶ __proto__: Array[0]
```

Figura 8.2: Tarefas vindas do servidor

Se você voltar na figura 8.2, vai perceber que o serviço nos envia também um valor `id` para cada uma das tarefas. Esse valor serve para que, ao alterarmos ou incluirmos uma tarefa, o banco de dados que está no servidor saiba de qual tarefa estamos falando. Como cada tarefa tem um `id` que não se repete, podemos ter certeza de que o servidor não vai apagar ou alterar uma tarefa errada.

Como no capítulo 6 nós ainda não estávamos pensando em gravar isso num banco de dados, vamos ter que alterar algumas funções do nosso script para que possamos lidar com esse valor `id`.

A alteração que faremos consiste em adicionar mais um elemento `div` dentro do item da tarefa. Como esse elemento `div` não deve ser visível ao usuário, vamos adicionar as linhas abaixo no final do nosso *CSS*.

```
.tarefa-id {
  visibility: hidden;
}
```

8.3 Usando parâmetros opcionais

Em seguida, vamos alterar a função `addTarefa` para que ela permita que possamos informar o `id` da tarefa. Vamos fazer de uma forma que o parâmetro `id` seja opcional. Caso ele não seja informado, vamos considerar que o `id` é igual a zero,

indicando que a tarefa ainda não existe no banco de dados. Isso também é útil para que tenhamos que alterar o mínimo possível do código que já existe.

Quando você deixa de informar um parâmetro numa função, ele fica com o valor `undefined` que, no JavaScript, significa que a variável não teve valor nenhum atribuído. Falando assim soa bem óbvio, não?

Vamos verificar logo no início da função se o valor de `id` recebeu um valor ou não. O código para isso fica assim:

```
id = id || 0;
```

O operador `||` é um **OU** booleano (ou lógico, se preferir). É uma forma resumida e elegante de escrevermos o código abaixo:

```
if(id !=== undefined) {
  id = id;
} else {
  id = 0;
}
```

Você deve concordar comigo que o segundo código, além de maior e mais difícil de ler, ainda tem um `id = id` muito esquisito ali no meio. *"Se a variável tiver um valor que não seja `undefined`, continue com o mesmo valor como se nada tivesse acontecido".*

Apresentações feitas, a função vai ficar assim:

```
function addTarefa(text, id) {
  id = id || 0;

  var $tarefa = $("<div />")
                .addClass("tarefa-item")
                .append($("<div />")
                        .addClass("tarefa-id")
                        .text(id))
                .append($("<div />")
                        .addClass("tarefa-text")
                        .text(text))
                .append($("<div />")
                        .addClass("tarefa-delete"))
                .append($("<div />")
                        .addClass("clear"));
```

```
  $("#tarefa-list").append($tarefa);

  $(".tarefa-delete").click(onTarefaDeleteClick);

  $(".tarefa-item").click(onTarefaItemClick);
}
```

Vamos aproveitar a função `addTarefa` para incluir nossas tarefas, passando o `id` que recebemos na nossa requisição AJAX:

```
function loadTarefas() {
  $("#tarefa").empty();

  $.getJSON(server + "/tarefas", {usuario: meu_login})
    .done(function(data) {
      console.log("data: ", data);
      for(var tarefa = 0; tarefa < data.length; tarefa++) {
        addTarefa(data[tarefa].texto, data[tarefa].id);
      }
    });
}
```

Existem mais coisas que precisam ser mexidas para poder lidar com o `id`, mas vamos alterar o restante do código conforme a necessidade, para que você não tenha que lidar com um monte de código sem necessidade.

8.4 Alterando tarefas

Para alterar uma tarefa, vamos enviar o texto e o id da tarefa para o serviço `/tarefa`, no singular, usando o método `post` do jQuery.

O método `post` funciona exatamente como os já conhecidos métodos `get` e `getJSON`: o primeiro parâmetro indica o endereço do serviço que vamos acessar, o segundo contém os parâmetros que vamos enviar ao servidor e o terceiro, opcional, recebe um callback que será executado quando a requisição for concluída com sucesso. Os eventos `done` e `fail` também podem ser usados com o método `post`.

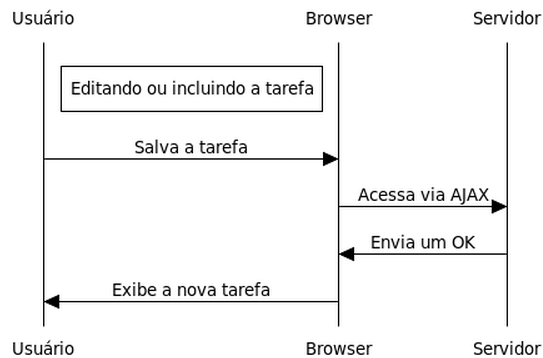



Figura 8.3: Alterando ou incluindo uma tarefa

Vamos criar uma função `updateTarefa`, que envia a tarefa atualizada para o servidor, que deve gravá-la num banco de dados para que possamos recuperá-la na próxima vez que entrarmos no gerenciador.

Essa função vai receber o texto e o id da tarefa como parâmetros. Como você só altera uma tarefa que já existe, e uma tarefa que já existe sempre deve ter um id, não faz sentido tratar o parâmetro `id` como opcional.

O serviço `/tarefa` recebe dois parâmetros: `tarefa_id`, que contém o id da tarefa a ser alterada, e `texto`, que contém o texto da tarefa.

```
function updateTarefa(text, id) {
  $.post(server + "/tarefa", {tarefa_id: id, texto: text});
}
```

Bem simples, não?

Alteraremos o nosso código já existente para que possamos usar essa nova função.

Primeiramente, vamos alterar a função `onTarefaItemClick` para permitir que o `id` seja mantido enquanto o usuário altera o texto da tarefa. Vamos fazer isso criando um elemento `div` que tenha a classe `tarefa-id`. Esse elemento vai ficar invisível para o usuário e vai guardar o valor do id da tarefa editada.

```
function onTarefaItemClick() {
  if(!$(this).is($lastClicked)) {
    if($lastClicked !== undefined) {
```

```
      savePendingEdition($lastClicked);
    }

    $lastClicked = $(this);

    var text = $lastClicked.children('.tarefa-text').text();
    var id = $lastClicked.children('.tarefa-id').text();

    var content = "<div class='tarefa-id'>" + id + "</div>" +
      "<input type='text' class='tarefa-edit' value='" +
      text + "'>";

    $lastClicked.html(content);

    $(".tarefa-edit").keydown(onTarefaEditKeydown);
  }
}
```

Vamos alterar também a função `savePendingEdition`, que é responsável por gerar o HTML do item da tarefa que estava sendo alterada e sumir com a caixa de texto usada na edição.

Também faremos com que essa função seja a responsável por enviar os dados para o servidor, através da função `updateTarefa`.

```
function savePendingEdition($tarefa) {
  var text = $tarefa.children('.tarefa-edit').val();
  var id = $tarefa.children('.tarefa-id').text();
  $tarefa.empty();
  $tarefa.append("<div class='tarefa-id'>" + id + "</div>")
        .append("<div class='tarefa-text'>" + text + "</div>")
        .append("<div class='tarefa-delete'></div>")
        .append("<div class='clear'></div>");

  updateTarefa(text, id);

  $(".tarefa-delete").click(onTarefaDeleteClick);

  $tarefa.click(onTarefaItemClick);
}
```

Agora as nossas tarefas já estão sendo carregadas automaticamente e gravadas no servidor assim que elas são alteradas. Para testar, faça o seguinte: altere uma

tarefa, grave-a e, em seguida, recarregue sua página. Você verá que o texto alterado da tarefa continuará sendo exibido na tela. Antes das nossas melhorias, todas as mudanças eram ignoradas e o valor original da tarefa voltava a ser exibido.

8.5 Uma introdução ao REST

Você já deve ter ouvido falar de REST em algum desses sites ou blogs que falam sobre arquitetura de software ou desenvolvimento para Web. Se nunca ouviu falar, recomendo que você corra atrás do tempo perdido para entender do que se trata.

REST é um assunto tão vasto e cheio de conteúdo que daria para encher um livro inteiro tranquilamente. A minha intenção aqui não é explicar REST em seus detalhes ou tornar o leitor um especialista no assunto. A ideia é que você tenha uma visão geral do que significa e, principalmente, entenda como isso se encaixa na nossa aplicação.

O padrão REST se tornou conhecido após Roy Fielding publicar sua dissertação de doutorado [5], onde explica os princípios e padrões para se desenvolver e utilizar uma aplicação usando REST.

Como você já deve saber, uma aplicação Web, e praticamente todo o resto dos sites do World Wide Web, comunica-se através de um protocolo chamado HTTP (Protocolo de transferência de Hipertexto). Esse protocolo usa duas partes distintas em suas chamadas: um **método** e um **recurso**.

Um **método** indica a ação a ser executada sobre o **recurso** indicado. Por isso pode ser chamado de **verbo**.

Já um **recurso** indica o arquivo, página ou serviço que deve ser acessado por um browser ou por uma aplicação.

Exemplificando, vamos entender o que acontece quando fazemos uma requisição ao serviço de consulta de CEPs, como fizemos no capítulo 7.

Nós usamos o método `get`, para acessar o endereço http://api.postmon.com.br/cep/04101300. Por baixo dos panos, o que o browser fez foi se conectar ao servidor que está no endereço `api.postmon.com.br`. De verdade mesmo, existe uma série de outros passos nesse meio tempo envolvendo servidores de nomes, mas isso fica como lição de casa para você.

Após se conectar no servidor, o browser vai enviar o comando:

```
GET /v1/cep/04101300
```

Ou seja, a requisição é feita pelo verbo `GET` e o recurso `/v1/cep/04101300`. É por isso que o método do jQuery se chama `get`. Da mesma forma, quando usamos

o método jQuery `post`, o que está sendo enviado para o servidor é o verbo `POST`.

http://api.postmon.com.br/cep/04101300
protocolo endereço do servidor recurso (URI)

Figura 8.4: As partes de um endereço

Existem outros verbos HTTP que são usados com REST além do GET e do POST. Quando estamos trabalhando com operações de cadastro (criar, ler, atualizar e deletar, conjunto de operações conhecido como **CRUD**), usamos quatro verbos HTTP:

- **GET** para ler
- **POST** para atualizar
- **PUT** para incluir
- **DELETE** para excluir

Como você vai perceber daqui a pouco, podemos usar o *mesmo recurso* para várias operações, desde que estejamos utilizando *verbos diferentes*.

REST não é apenas isso, mas para o que vamos precisar é mais do que suficiente.

8.6 Utilizando PUT e DELETE no browser

Apesar do jQuery permitir o uso de PUT e DELETE, a maioria dos navegadores aceita apenas os protocolos GET e POST em suas requisições, o que nos obriga a usar alguns truques para desenvolver aplicações usando REST que funcionem em qualquer browser.

Boa parte das bibliotecas atuais usadas para desenvolver aplicações Web conseguem entender que você está querendo usar PUT ou DELETE se você informar isso num parâmetro especial chamado `_method` e fizer suas requisições usando `POST`. Os nossos serviços foram feitos com Rails, que segue essa prática.

8.7 Adicionando tarefas com REST

O processo de adicionar tarefas não é muito diferente do que fizemos para trabalhar com alterações, com a mudança de que a tarefa ainda não existe no banco de dados,

e o id será zero antes de enviarmos para o servidor, e receberá um novo valor assim que ele for gravado no banco de dados. Isso vai nos permitir que a tarefa seja alterada e gravada normalmente como qualquer outra que vimos até agora.

Vamos criar uma função chamada newTarefa, que vai receber como parâmetro o texto da tarefa e um objeto jQuery contendo o elemento div que contém o id. Assim, a função vai poder atualizar o valor do id assim que receber a resposta do servidor. Lembre-se de que estamos trabalhando com processos assíncronos e não sabemos quanto tempo vai levar para essa resposta chegar ao browser.

Conforme vimos anteriormente, não podemos nos esquecer de informar o verbo HTTP que vamos usar. Nesse caso, como se trata de uma inclusão, usaremos PUT.

```
function newTarefa(text, $div) {
  $.post(server + "/tarefa",
    {usuario: meu_login,
      texto: text,
      _method: "PUT"})
    .done(function(data) {
      $div.text(data.id);
    });
}
```

Vamos alterar a função addTarefa para que possamos enviar a nova tarefa para o servidor. Adicionaremos uma verificação na função para termos certeza de que apenas tarefas com id igual a zero sejam incluídas, pois estamos usando essa mesma função para adicionar na tela as tarefas que vieram do servidor ao carregarmos a lista.

```
function addTarefa(text, id) {
  id = id || 0;

  var $tarefa = $("<div />")
                .addClass("tarefa-item")
                .append($("<div />")
                        .addClass("tarefa-id")
                        .text(id))
                .append($("<div />")
                        .addClass("tarefa-text")
                        .text(text))
                .append($("<div />")
                        .addClass("tarefa-delete"))
```

```
                .append($("<div />")
                        .addClass("clear"));

  $("#tarefa-list").append($tarefa);

  $(".tarefa-delete").click(onTarefaDeleteClick);

  $(".tarefa-item").click(onTarefaItemClick);

  if(id === 0) {
    var div = $($tarefa.children(".tarefa-id"));
    console.log("id", div);
    newTarefa(text, $(div));
  }
}
```

Inclua algumas tarefas e carregue novamente a página para testar.

8.8 Excluindo tarefas

Finalmente, vamos aprender a excluir uma tarefa. Vamos alterar a função `onTarefaDeleteClick`. Tudo o que precisamos saber é o id da tarefa e enviá-lo para o servidor.

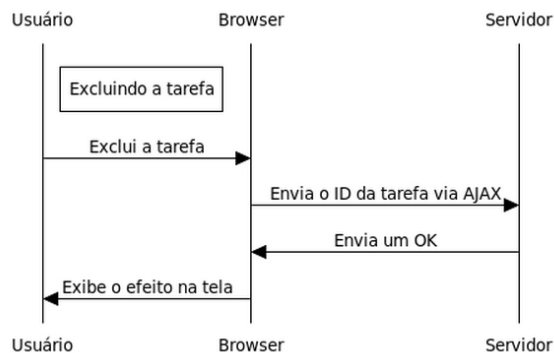



Figura 8.5: Excluindo uma tarefas

Aqui também vamos usar o método `post`, do jQuery, e o verbo HTTP `delete`, passando o id do elemento através do parâmetro `tarefa_id`. Fique atento para

informar o id da tarefa **antes** de excluir o elemento do DOM, ou então você vai ficar sem saber o que foi excluído.

Nosso código vai ficar assim:

```
function onTarefaDeleteClick() {
  $(this).parent('.tarefa-item')
    .off('click')
    .hide('slow', function() {
      $this = $(this);

      $.post(server + "/tarefa",
        {usuario: meu_login,
          tarefa_id: $this.children(".tarefa-id").text(),
          _method: "DELETE"});

      $(this).remove();
    });
}
```

Vamos excluir uma tarefa e recarregar a página. Você vai perceber que ela foi realmente excluída.

Finalmente terminamos de desenvolver o nosso gerenciador de tarefas.

A essa altura, você já deve ser capaz de desenvolver suas próprias aplicações com jQuery e entender o código de aplicações feitas por outras pessoas, mas ainda temos um bom caminho pela frente até que você conheça todos os recursos que o jQuery pode oferecer.

No próximo capítulo vamos apresentar o jQuery UI, que é uma biblioteca de componentes visuais que trabalha sobre o jQuery.

Capítulo 9

jQuery UI

"Se precisa de um 'jeitinho' de usar, é porque a interface com o usuário está quebrada."
– Douglas Anderson

9.1 Usando o jQuery UI

A biblioteca jQuery UI foi criada para permitir a rápida criação e utilização de componentes visuais, efeitos e temas, aproveitando toda a facilidade que o jQuery traz em relação a diferentes tipos e versões de navegadores.

Existem duas formas de fazer o download da biblioteca para o seu projeto, conforme podemos ver na figura 9.1.

Figura 9.1: Essas cores ficaram estranhas...

A primeira é clicar no link *Stable*, que está sob a inscrição *Quick Downloads*. Essa opção é muito boa para quem está começando e quer experimentar todos os recursos que a biblioteca oferece. Por outro lado, a biblioteca inteira do jQuery UI é grande, e pode deixar a nossa aplicação bem pesada quando for acessada por uma conexão lenta.

Ao lado de *Stable* (estável), você tem opção *Legacy* (legado), que permite que você faça download da versão anterior e desatualizada da biblioteca. A menos que você tenha um bom motivo para precisar dessa versão, ignore-a solenemente.

A segunda forma ajuda a resolver o problema do excesso de peso da biblioteca. Clicando em *Custom Download* (download personalizado), você será direcionado para uma ferramenta, chamada *Download Builder*, que permite que você baixe apenas o que é necessário, tornando a biblioteca menor e, consequentemente, mais rápida de ser carregada. Sempre que eu preciso usar jQuery UI em uma aplicação, gero um download específico para as necessidades do projeto. Você pode acessar esse download personalizado também pelo link:

http://jqueryui.com/download/

No Download Builder, você pode selecionar os módulos, componentes e efeitos que deseja utilizar. É possível também selecionar o tema que será usado para alterar o visual dos componentes. É possível também criar um módulo de acordo com a identidade visual da sua aplicação.

Após o download, basta incluir no seu HTML o arquivo JavaScript que contém o jQuery UI. Lembre-se de **sempre incluir o jQuery UI *depois* do jQuery**.

```
<script src="js/jquery-1.9.0.js"></script>
<script src="js/jquery-ui-1.10.0.custom.js"></script>
```

Obviamente, o local onde estão seus arquivos JavaScript e seus respectivos nomes podem ser ligeiramente diferentes.

9.2 As diversas partes do jQuery UI

O jQuery UI é formado de quatro módulos, que podem ser selecionados automaticamente pelo Download Builder conforme você escolhe os componentes que vai utilizar.

UI Core

Contém o núcleo do jQuery UI. Todos os demais componentes dependem desta parte.

Interactions

É a parte que dá suporte a coisas como *drag-and-drop* (quando você arrasta e solta algum componente pela tela), ordenar itens numa tabela usando o mouse e mesmo mudar o tamanho de algum elemento da tela se você desejar.

Widgets

Widgets são componentes visuais que já vêm com o jQuery UI. Dentre os muitos componentes disponíveis, você tem acesso a menu, abas, botões e janelas que só existem dentro da sua página.

Vamos dar mais atenção aos *widgets* mais para frente.

Effects

O jQuery já vem com alguns efeitos simples, mas o jQuery UI traz treze efeitos novos para que você possa impressionar o usuário.

Como fica difícil demonstrar os efeitos em um livro, recomendo que você acesse a página de demonstração do jQuery UI no endereço abaixo:

http://jqueryui.com/demos/

9.3 Temas

Por melhor que seja a sua aplicação, ela estará fadada ao fracasso se tiver um visual desagradável. Para ajudar a deixar o seu design com aparência consistente e profissional, o jQuery UI permite o uso de temas.

Temas são conjuntos de fontes e cores que são aplicadas em todos os componentes criados pelo jQuery UI, e podem ser trocados e criados conforme a sua necessidade, fazendo com que toda a sua aplicação troque de aparência de uma vez só.

Abaixo temos um exemplo de diferentes temas aplicados a uma mesma tela, conforme você pode observar nas figuras 9.2 e 9.3.

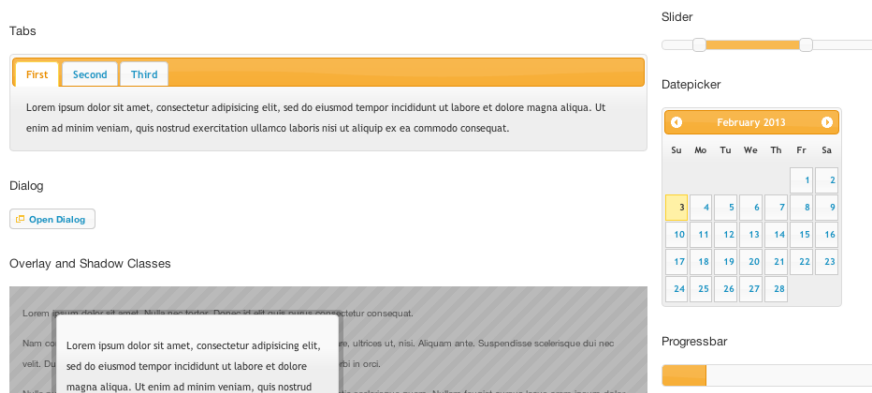



Figura 9.2: Tema UI Lightness

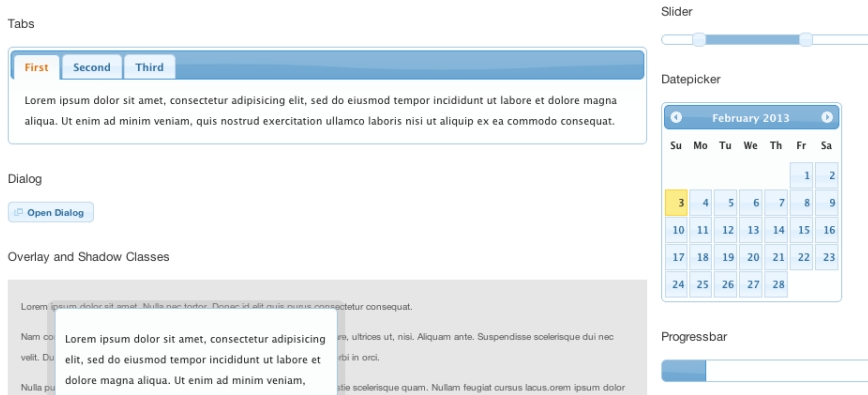



Figura 9.3: Tema Redmond

Você pode experimentar e criar temas usando uma ferramenta chamada Theme Roller, que está no endereço:

http://jqueryui.com/themeroller/

Selecione a aba *Roll Your Own*, algo como Crie seu próprio tema, ou *Gallery* caso queira experimentar temas já existentes.

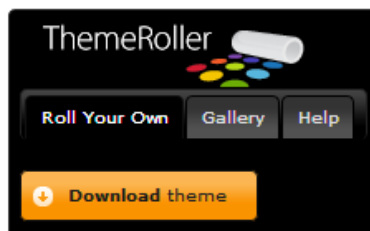



Figura 9.4: Detalhe do Theme Roller

9.4 Organizando seus dados com accordion

Accordion, ou sanfona se você traduzir porcamente, é um componente que se tornou popular com o Microsoft Office, e exibe uma parte do conteúdo enquanto oculta todas as outras partes. É muito usado para exibir um texto por vez ou um conjunto de componentes de acordo com um determinado assunto ou função.

Para utilizar o *accordion*, você precisa definir um elemento *div* que vai conter o componente e utilizar elementos *h3* para definir os títulos das seções, que terão o conteúdo também dentro de elementos *div*.

No código abaixo nós vamos montar uma pequena página para a empresa *Zeca Urubu S/A*. Eu sinceramente recomendo que você não compre nada que tenha a marca deles. O código completo pode ser encontrado em:

http://jsfiddle.net/pbalduino/A2zxZ/

```
<div id="accordion">
  <h3>Quem somos</h3>
  <div>
    <p>Somos uma empresa que escreve aquelas coisas bonitas de
    visão e missão, mas fazemos diferente na vida real.</p>
  </div>
  <h3>O que fazemos</h3>
  <div>
    <p>Criamos produtos que são vendidos com um ano de garantia e
    quebram após um ano e meio de uso.</p>
  </div>
  <h3>Trabalhe conosco</h3>
  <div>
    <p>Se você é do tipo que venderia um gato morto para a própria
    avó, venha trabalhar conosco!
    </p>
  </div>
</div>
```

O código JavaScript será esse abaixo. Considere que você já tenha adicionado os arquivos do jQuery e do jQuery UI no cabeçalho do HTML.

```
$(function() {
  $("#accordion").accordion();
});
```

Na figura 9.5 podemos ver como ficaria essa página.

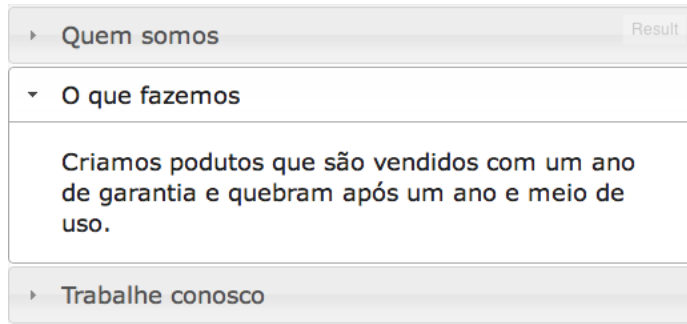



Figura 9.5: Página da Zeca Urubu S/A

Você pode encontrar a documentação desse componente no endereço abaixo:

http://jqueryui.com/accordion/

9.5 Auto-completando

Um recurso muito bacana que ficou bem conhecido com o surgimento da Web 2.0 é o de auto-completar um texto.

Para que o recurso funcione, você precisa informar ao jQuery UI quais são os valores a serem exibidos na lista que aparece abaixo da caixa de texto. Você pode fazer isso passando uma lista pronta, como fizemos no exemplo, ou usando o que já aprendemos até aqui e carregando dinamicamente via AJAX.

O exemplo funcional pode ser encontrado no link abaixo:

http://jsfiddle.net/pbalduino/srN2f/

Para o HTML, vamos precisar apenas

```
<div class="ui-widget">
  <label for="linguagens">Linguagem: </label>
  <input id="linguagens" />
</div>
```

```
var linguagens = [
    "Clojure",
    "Common LISP",
    "Erlang",
    "F#",
    "Haskell",
```

```
    "JavaScript",
    "OCaml",
    "Scala",
    "Scheme"
    ];

$("#linguagens").autocomplete({source: linguagens});
```

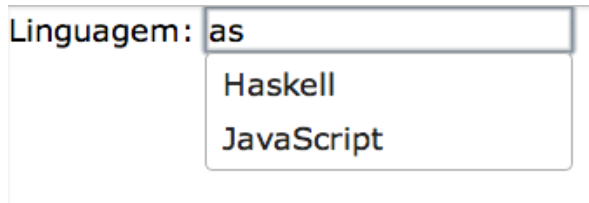



Figura 9.6: Autocomplete simples

A documentação completa desse componente está em:
http://jqueryui.com/autocomplete/
http://jsfiddle.net/pbalduino/NFprU/

9.6 Usando botões mais bonitos

O jQuery UI também permite que você deixe os botões mais bonitos. É o tipo de situação em que você vai usar um canhão para matar uma mosca, mas não custa mostrar como fica.

O código completo pode ser encontrado em:
http://jsfiddle.net/pbalduino/RxM94/

```
<button class="chique">Um botão bonito</button>
<button>Um botão normal</button>

<hr />

<input type="submit" class="chique" value="Um submit bonitão">
<input type="submit" value="Um submit normal">

<hr />
```

```
<a href="#" class="chique">Um link em forma de botão</a>
<a href="#">Um link qualquer</a>
```

O código JavaScript também é bem simples:

```
$(function() {
  $(".chique").button();
});
```

Podemos ver na figura 9.7

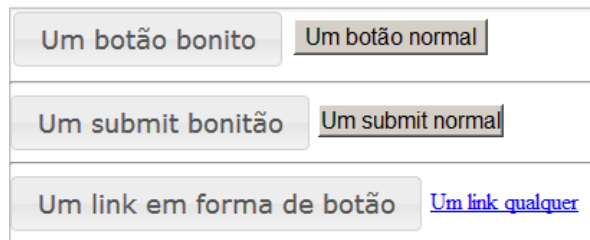



Figura 9.7: Botões elegantes

Existem várias outras opções de botões. Você pode saber mais em:
http://jqueryui.com/button/
Experimente aplicar esses novos botões na nossa aplicação de tarefas!

9.7 Escolhendo a data com o calendário

Um grande problema para quem escreve código para o *back-end*, que é a parte da sua aplicação que roda no servidor, é garantir que a data informada pelo usuário esteja num formato correto e seja válida.

Por padrão, uma caixa de texto aceita qualquer tipo de texto, permitindo inclusive que o usuário escreva o próprio nome no lugar da data de nascimento.

O HTML5 resolve parcialmente esse problema adicionando um novo tipo de campo `input`, específico para datas. Podemos usá-lo com o código abaixo:

```
<label for="nascimento">Data de nascimento : </label>
<input id="nascimento" type="date" value="2010-02-08"/>
```

Teremos um resultado como o da figura 9.8. De acordo com a documentação oficial[2], não temos como formatar a data, além de não termos garantia nenhuma de que o browser do usuário suporta esse novo elemento. Até a data em que escrevi isso, apenas o Chrome 20, ou mais novo, e o Opera 11, ou mais novo, exibiam corretamente o campo data. O Firefox na versão 18.0.2, o Safari 6 e o Internet Explorer ainda não reconhecem o tipo `date`.

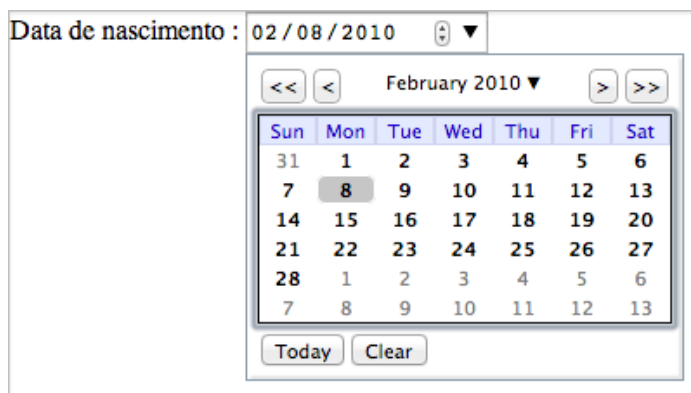



Figura 9.8: Campo Date do HTML5

Você pode testar o uso do componente no seu browser no link abaixo:
http://jsfiddle.net/pbalduino/fz6Lk/

Para termos certeza de que o browser do usuário vai exibir corretamente o calendário, vamos usar o componente **DatePicker** do jQuery UI, que é um calendário altamente configurável e com visual integrado ao tema que você escolher.

Vamos abusar das opções de configuração para que você possa ver como deixar o calendário totalmente em português e com o nosso formato de data.

Novamente, basta declarar uma caixa de texto no HTML. Vamos atribuir `nascimento` ao `id`.

Caso você tenha se perguntado, esse elemento `div` com a classe `ui-widget` faz com que o seu conteúdo obedeça à formatação do tema escolhido.

```
<div class="ui-widget">
    <label for="nascimento">Data de nascimento:</label>
    <input id="nascimento" type="text">
</div>
```

Para o código JavaScript, vamos usar o método `datepicker`, que recebe um objeto contendo as configurações, que são totalmente opcionais. Claro que não vai ficar tão legal se você não passar configuração nenhuma, mas vai funcionar de qualquer maneira.

```
$(function () {
    $("#nascimento").datepicker({
        showWeek: true,
        firstDay: 0,
        dateFormat: "dd/mm/yy",
        dayNames: ["Domingo",
            "Segunda-Feira",
            "Terça-Feira",
            "Quarta-Feira",
            "Quinta-Feira",
            "Sexta-Feira",
            "Sábado"],
        dayNamesMin: ["D", "S", "T", "Q", "Q", "S", "S"],
        monthNames: ["Janeiro",
            "Fevereiro",
            "Março",
            "Abril",
            "Maio",
            "Junho",
            "Julho",
            "Agosto",
            "Setembro",
            "Outubro",
            "Novembro",
            "Dezembro"],
        monthNamesShort: ["Jan",
            "Fev",
            "Mar",
            "Abr",
            "Mai",
            "Jun",
            "Jul",
            "Ago",
            "Set",
            "Out",
            "Nov",
```

```
            "Dez"],
        showButtonPanel: true,
        currentText: "Hoje",
        closeText: "Fechar",
        weekHeader: "#"
    });
});
```

Note que no código pedimos para adicionar uma coluna com o número da semana, traduzimos todos os dias da semana e meses do ano, nas formas longa e curta, mandamos exibir dois botões na parte de baixo, sendo que um deles move o calendário para o dia de hoje, e dissemos também que a semana começa no Domingo (use o número 1 se você quiser que comece na Segunda-Feira).

Podemos ver o resultado na figura 9.9.

Figura 9.9: Calendário

Você pode verificar todas as opções do DatePicker no endereço abaixo:

http://api.jqueryui.com/datepicker/

9.8 Exibindo janelas dentro da janela

É possível exibir pequenas janelas dentro do browser, chamadas de *Dialogs*, ou *caixas de diálogo*. Essas Dialogs são úteis para exibir mensagens do sistema, confirmar ações

ou mesmo para exibir diferentes processamentos ao mesmo tempo, dependendo da sua necessidade e criatividade.

Vamos usar como exemplo uma mensagem do sistema que exige que o usuário escolha entre duas opções. Primeiro vamos criar um elemento `div` que vai guardar todo o conteúdo da Dialog. Todo o texto, imagem ou qualquer outra informação que você quer que apareça, estará dentro desse `div`.

```
<div id="dialog" title="Aviso do sistema">
  <p>Você acha que esse é o melhor
  livro de JavaScript que já existiu?</p>
</div>
```

Em seguida, vamos usar o método `dialog` para configurar a Dialog, passando um objeto contendo as configurações. No nosso caso, vamos informar que temos dois botões, cada um executando seu próprio callback. Para manter o código legível e fácil de entender, eu separei as configurações numa variável `opcoes`, e criei uma função para cada callback.

O nosso código vai ficar assim:

```
$(function () {
  function onSimClick() {
    alert("Uhu! Você escolheu 'Sim'!");
    $(this).dialog("close");
  };

  function onClaroClick() {
    alert("\\o/");
    $(this).dialog("close");
  };

  var opcoes = {
    buttons: {
      "Sim": onSimClick,
      "Claro!": onClaroClick
    }
  };

  $("#dialog").dialog(opcoes);
});
```

O código completo pode ser encontrado no endereço abaixo:

http://jsfiddle.net/pbalduino/s4X4B/

Veremos a mensagem como na figura 9.10

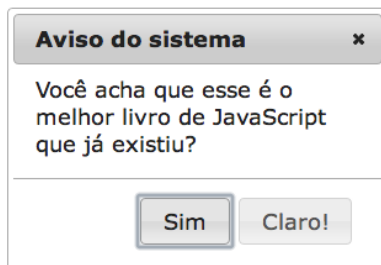



Figura 9.10: Aí o usuário aperta ESC =(

A documentação completa desse componente pode ser consultada no endereço abaixo:

http://api.jqueryui.com/dialog/

9.9 O problema com o jQuery UI

Você já deve ter ouvido alguém criticar o jQuery e o jQuery UI por *serem muito pesados*.

De fato, se você baixar a biblioteca completa com algum tema para a sua aplicação, você pode engordá-la em até 300KB desnecessariamente.

Se você estiver pensando apenas em usuários com banda larga, navegadores modernos e caches que funcionam, o problema todo desaparece após um ou dois segundos a mais na carga da aplicação.

Porém, se você pensar que em nosso país a banda larga não é tão larga assim, que a quantidade de usuários que acessam a Internet através de dispositivos móveis está aumentando consideravelmente a cada mês e que a qualidade e a velocidade da conexão oferecida pelas empresas de telefonia é uma piada de mau gosto, chegaremos à conclusão de que quanto menos, melhor.

Se você não tem necessidade de utilizar temas da maneira como o jQuery UI apresenta, eu recomendo que você entre em http://plugins.jquery.com/ e procure por alternativas mais leves para o que você precisa.

Em alguns casos extremos, é recomendável que você repense a interface com o usuário para que ela fique leve e fácil de usar. Será mesmo que você precisa de uma

tela carregada de componentes e opções, como se fosse um Microsoft Word com todas as barras de ferramentas, ou você poderia apresentar as mesmas informações de uma forma mais inteligente e simplificada?

Já que falamos em dispositivos móveis, vamos falar rapidamente sobre jQuery Mobile no próximo capítulo.

Capítulo 10

jQuery Mobile

"Não use. Simples assim."
– Sérgio Lopes

10.1 Enfrentando o mobile

De acordo com o *Cisco Visual Networking Index*[1], um relatório publicado anualmente pela Cisco, em 2012 presenciamos um aumento de 81% no uso de smartphones. De acordo com o mesmo relatório, o volume trafegado por smartphones no mesmo ano foi oito vezes maior do que o volume de toda a Internet no ano 2000.

Há algum tempo os dispositivos móveis deixaram de ser figurantes na Internet para se tornar parte do dia a dia de desenvolvedores de aplicações Web, empresas e usuários.

Atentos a isso, os desenvolvedores do jQuery lançaram em Novembro de 2011 a primeira versão estável do jQuery Mobile, visando atender ao crescente mercado de Internet móvel.

Para quem já utilizou o jQuery UI ou acompanhou o capítulo anterior, o jQuery Mobile não vai causar nenhuma surpresa.

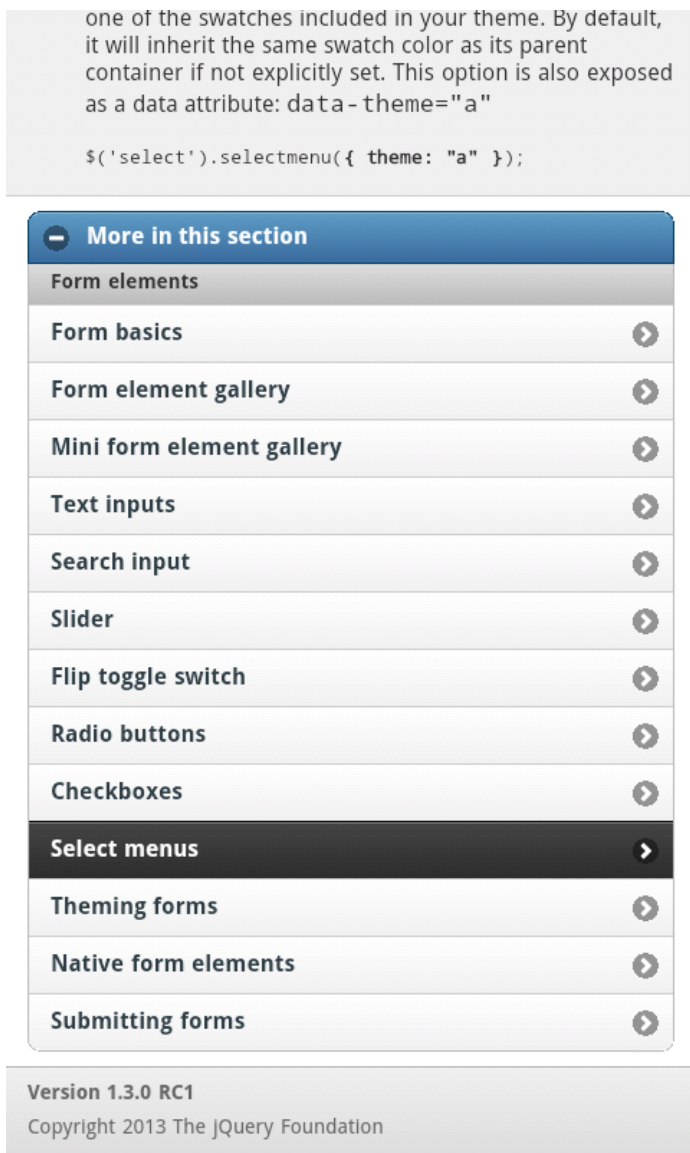



Figura 10.1: jQuery Mobile acessado de um celular

Ao contrário do que acontece com o jQuery puro em relação aos navegadores

mais utilizados, o jQuery Mobile não garante que sua aplicação ou site vão ser executados corretamente em todos os dispositivos. Você pode acessar o endereço abaixo para verificar quais dispositivos são compatíveis com a biblioteca:

http://jquerymobile.com/gbs/

A página inicial do jQuery Mobile (http://jquerymobile.com/) oferece uma ferramenta para que você possa experimentar os diversos componentes em um simulador de dispositivo móvel.

Assim como acontece com o jQuery UI, o jQuery Mobile também tem o *Theme Roller*, para que você crie seus próprios temas e deixe a aplicação com as cores que você quiser.

O Theme Roller para jQuery Mobile está em http://jquerymobile.com/themeroller/

10.2 Cuidado com o tempo de carregamento

Caso você esteja utilizando uma aplicação empacotada com uma ferramenta como o PhoneGap, o tempo de carregamento não é algo preocupante, uma vez que todos os seus arquivos já estarão instalados no seu aparelho. O problema existe, e é grave, quando você estiver usando jQuery Mobile para desenvolver o seu site.

O jQuery Mobile também oferece uma opção para que você faça o download apenas daquilo que realmente for necessário. Acesse o link abaixo para experimentar:

http://jquerymobile.com/download-builder/

Só que isso ainda não resolve o problema. Por mais enxuta que seja a sua instalação do jQuery Mobile, você ainda é obrigado a carregar todo o peso do jQuery, que ainda não é modularizado e contém quilos de código de que, possivelmente, você nunca vai precisar.

Sérgio Lopes, instrutor da Caelum, escreveu um artigo muito bom, no blog da Caelum, sobre os pontos negativos de se utilizar jQuery Mobile[13]. Tomei a liberdade de citar o trecho desse artigo em que ele apresenta, de forma concisa, um bom motivo para não usarmos jQuery em sites para dispositivos móveis.

"[Com o jQuery] são 32 KB de dados gzipados para se transferir numa rede potencialmente ruim como 3G. E quando os dados chegam, ocupam 95 KB sem gzip (mesmo minificado). E tudo isso tem que ser lido e parseado antes mesmo de se pensar em executar seu código. As estatísticas de carregamento do jQuery em

dispositivos móveis são assustadoras. Num iPad 2 topo de linha, só o parsing e eval do jQuery demora 5x mais que no Desktop. Um iPhone 3, que, apesar de antigo, é melhor que a maioria dos celulares que as pessoas têm em seu bolso, demora incríveis 850ms. E mesmo um iPhone 4 gasta uma eternidade se comparado ao Desktop – 320ms vs. 35ms."
– Sérgio Lopes

Como alternativa, podemos usar uma ferramenta chamada Zepto.js, que pode ser encontrada no endereço abaixo:

http://zeptojs.com/

Praticamente tudo o que você aprendeu sobre seletores, callbacks, AJAX e eventos pode ser aplicado no ZeptoJS, com a vantagem de ter um arquivo quatro vezes menor para baixar através de uma rede problemática (90.5 KB do jQuery 1.9.1 contra 23.5 do Zepto 1.0rc1).

É claro que não existe bala de prata e que, em alguns casos o Zepto simplesmente não vai ser suficiente para substituir o jQuery.

Apesar de você já ter lido isso antes, uma saída mais efetiva é repensar a sua interface, tornando-a mais simples e direta, utilizando JavaScript puro onde for possível.

Quando não for possível se livrar do uso uma biblioteca, experimente o uso de *microframeworks* como o **Hammer.js**, disponível em http://eightmedia.github.com/hammer.js/, ou algum da lista do **microjs**, disponível em http://microjs.com/.

Independente da ferramenta que você queira utilizar, tenha sempre em mente que o desenvolvimento para Web para dispositivos móveis é muito semelhante ao que tínhamos há dez ou quinze anos, quando as pessoas utilizavam discadores em linhas telefônicas e simplesmente não podiam, e não queriam, esperar dez segundos para que uma página fosse carregada.

Capítulo 11

Orientação a objetos no JavaScript

"JavaScript é uma linguagem criada sem muito cuidado, mas dentro dela há uma linguagem elegante e muito melhor."
– Douglas Crockford

11.1 Objetos no JavaScript

Apesar de JavaScript ser uma linguagem orientada a objetos, o conceito de objeto é um pouco diferente do que você pode estar acostumado, em especial se veio de linguagens como Java, C#, C++, Ruby, Python ou outra largamente difundida no mercado.

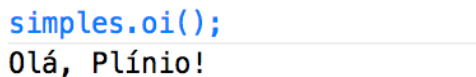
Primeiramente, não existe o conceito de *classes* no JavaScript tal como o conhecemos nas linguagens que citei, então você não precisa de uma classe para criar seus objetos.

Em segundo lugar, o que normalmente tratamos como objetos no JavaScript não são necessariamente objetos, mas funções com propriedades. E os objetos que são realmente objetos são diferentes dos objetos de outras linguagens.

Agora que eu confundi o que você já sabia, vou mostrar passo a passo que não é nada disso.

Um objeto simples

O jeito mais simples de criarmos um objeto em JavaScript é utilizando um *dicionário*, ou se preferir, mapa ou hashtable. Nós utilizamos objetos assim durante os capítulos em que falamos sobre JSON e AJAX, e também para passarmos configurações para os métodos do jQuery Mobile.

O objeto mais simples que podemos criar é simplesmente o `{ }`. Ele não faz nada, nem guarda valor nenhum, mas podemos adicionar funcionalidades nele conforme as nossas necessidades.

```
var simples = { };

simples.nome = "Plínio";

simples.oi = function() {
  console.log("Olá, " + this.nome + "!");
}

simples.oi();
```

O resultado pode ser visto na figura 11.1.

```
simples.oi();
Olá, Plínio!
```

Figura 11.1: Oi!

Perceba que, exatamente por estarmos lidando com um dicionário, podemos adicionar propriedades e métodos ao nosso objeto a qualquer momento. Para o JavaScript, um objeto é apenas um dicionário contendo chaves e valores.

Podemos reescrever o código acima da forma a seguir, o que parece mais familiar para quem vem de linguagens com orientação a objetos baseadas em classes:

```
var outro_simples = {
  nome: "Plínio",
```

```
  oi: function() {
    console.log("Olá, " + this.nome + "!");
  }
};

outro_simples.oi();
```

Perceba que, nos dois casos, foi necessário usar `this` para acessarmos o conteúdo de `nome`. Isso indica que devemos procurar, dentro do objeto atual, por uma variável ou propriedade chamada `nome`. Se você não usar `this`, o JavaScript vai procurar por uma variável chamada `nome`, o que não existe no nosso código.

Da forma como criamos o objeto, é possível também que o nome seja alterado, bastando acessar `outro_simples.nome`, como vemos na figura 11.2. Se quisermos proteger o valor de `nome` e não permitir que o usuário altere o valor da propriedade, teremos problemas, já que a sintaxe do dicionário não permite que criemos uma variável ali dentro.

```
> outro_simples.nome = "Jeremias";
  "Jeremias"
> outro_simples.oi();
  Olá, Jeremias!
```

Figura 11.2: Um 'olá mundo' mais bacana

Outro problema que podemos encontrar usando essa abordagem é que não há uma maneira de termos dois objetos do mesmo tipo que mantenham seus valores isolados sem repetir quase todo o código.

Imagine que temos dois funcionários. Cada um tem seu próprio nome, cargo e salário. Vamos tentar implementar isso com o que aprendemos sobre objetos até agora.

```
var paulo = {
  nome: "Paulo",
  cargo: "Analista de Sistemas",
  salario: 5000
};
```

Ao exibirmos o nome do objeto `paulo`, teremos a saída já esperada na figura 11.3.

```
> paulo.nome;
  "Paulo"
```

Figura 11.3: O nome de paulo é Paulo

Agora imagine que o usuário não quer repetir todo o código usado para criar o objeto `paulo` e tente fazer uma cópia para o objeto `pedro`. Na cabeça dele, cada objeto continuará sendo único e assim, ele poderá fazer quantas cópias forem necessárias para que todos os funcionários da empresa sejam mapeados.

Então o usuário faz o seguinte:

```
var pedro = paulo;

pedro.nome = "Pedro";
pedro.cargo = "Gerente de contas";
pedro.salario = 4500;
```

Ao exibir o valor de `nome` de `pedro`, o usuário recebeu "Pedro" de volta. Excelente, não?

Mas, ao exibir o valor de `nome` de `paulo`, ele teve uma surpresa como a da figura 11.4.

```
> pedro.nome;
  "Pedro"
> paulo.nome;
  "Pedro"
```

Figura 11.4: O nome de pedro é Pedro. E o de paulo?

Claro! Isso acontece porque a `var` guarda uma referência ao objeto, e não o objeto em si.

Mas e se quiséssemos criar um novo objeto que já possua as mesmas funções e atributos de outro objeto? Isso é, como fazer algo parecido com uma classe?

11.2 Funções para quem não tem classe

Em JavaScript não existe o conceito de classes, mas as funções podem muito bem assumir esse papel, o que significa que podemos criar objetos a partir de funções.

Vamos reescrever o nosso primeiro exemplo, o do *"Olá, fulano!"*, usando uma função ao invés de um dicionário.

```
function OlaSimples() {
  this.nome = "Plínio";

  this.oi = function() {
    console.log("Olá, " + this.nome + "!");
  }
}
```

Vamos instanciar um objeto `simples` para demonstrar como podemos usar esse objeto. A essa função usada para criar um objeto chamamos de **Construtor**. Note a semelhança com o Java, C# ou C++:

```
var simples = new OlaSimples();

simples.oi();
```

```
> simples.oi();
Olá, Plínio!
```

Figura 11.5: Um olá com cara de orientação a objetos

Se quisermos alterar o nome, ainda podemos acessar a propriedade `nome`. Porém, ao contrário do exemplo anterior, podemos esconder `nome` para que seu valor não possa ser alterado. Para isso, devemos declarar a propriedade `nome` como uma variável.

```
function OlaSimples(seuNome) {
  var nome = seuNome;

  this.oi = function() {
    console.log("Olá, " + nome + "!");
  }
```

```
}

simples = new OlaSimples("Plínio");

simples.oi();
```

Na figura 11.6 você pode ser o resultado do nosso código. Agora o nome só pode ser definido na criação do objeto. A essa propriedade de podermos guardar as informações dentro de um objeto e, eventualmente, escondê-las do mundo exterior, chamamos de **encapsulamento**, e é um dos conceitos fundamentais da orientação a objetos.

```
simples = new OlaSimples("Plínio");

simples.oi();
Olá, Plínio!
```

Figura 11.6: Um olá com parâmetro no construtor

Uma característica do encapsulamento é que cada objeto guarda suas próprias informações, fazendo com que a alteração em um deles seja invisível aos demais. Vamos usar o exemplo dos funcionários para exemplificar.

Primeiro, vamos criar um construtor para os objetos que simbolizam os empregados. Não vamos nos preocupar com o fato de podermos alterar os dados do funcionário após criarmos o objeto, pois agora você sabe como alterar o construtor para que as informações sejam alteradas somente quando você quiser:

```
function Funcionario(nome, cargo, salario) {
  this.nome = nome;
  this.cargo = cargo;
  this.salario = salario;
}

var paulo = new Funcionario("Paulo", "Analista de sistemas", 5000);
var pedro = new Funcionario("Pedro", "Gerente de contas", 4500);

console.log("O nome de paulo é " + paulo.nome);
console.log("O nome de pedro é " + pedro.nome);
```

Como podemos ver na figura 11.7, cada objeto tem seus próprio valores, e um não influencia no outro, o que é exatamente o que esperamos ao lidar com objetos.

```
console.log("O nome de paulo é " + paulo.nome);
console.log("O nome de pedro é " + pedro.nome);
O nome de paulo é Paulo
O nome de pedro é Pedro
```

Figura 11.7: paulo é Paulo, pedro é Pedro

Note que não foi preciso repetir código para criarmos dois objetos distintos, mas do mesmo tipo.

11.3 Entendendo prototipação

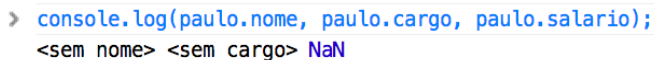
O JavaScript é uma linguagem que faz uso de **prototipação**, o que significa que, ao invés de termos uma classe herdando características de outra classe, os objetos herdam suas características de outros objetos.

Vamos usar o exemplo dos funcionários para exemplificar como usar prototipação usando JavaScript.

Criaremos um construtor Funcionario, sem nenhuma propriedade ou método. Conforme o esperado, se tentarmos acessar o nome, cargo ou salário, que ainda não existem, receberemos `undefined`, como podemos ver na figura 11.8

```
function Funcionario() {
};

var paulo = new Funcionario();

console.log(paulo.nome, paulo.cargo, paulo.salario);
```

```
console.log(paulo.nome, paulo.cargo, paulo.salario);
undefined undefined undefined
```

Figura 11.8: O funcionário anônimo

Usando o código acima, vamos usar prototipação para adicionar características a objetos que já existem.

```
Funcionario.prototype.nome = "<sem nome>";
Funcionario.prototype.cargo = "<sem cargo>";
Funcionario.prototype.salario = NaN;
```

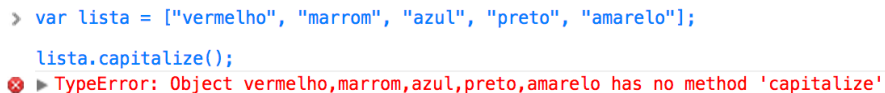
Exibiremos novamente o que o funcionário `paulo` tem:

```
console.log(paulo.nome, paulo.cargo, paulo.salario);
```

Agora as propriedades de `Funcionario` existem, como podemos ver na figura 11.9, e contêm valores padrão que você pode alterar conforme suas necessidades.

```
> console.log(paulo.nome, paulo.cargo, paulo.salario);
  <sem nome> <sem cargo> NaN
```

Figura 11.9: O funcionário padrão

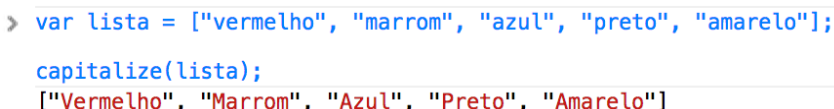
Repare que usando `prototype` você pode adicionar características em todos os objetos que forem criados com o construtor `Funcionario` e mesmo nos que já haviam sido criados.

Aqui você deve estar se perguntando qual a vantagem dessa tal prototipação. Uma das principais vantagens é a possibilidade de poder adicionar características a classes já existentes, fazendo com que seu código fique mais limpo e reutilizável.

Vamos imaginar que você queira que cada nome de uma lista tenha a primeira letra em maiúscula e as demais minúsculas. Vamos ver se o JavaScript já nos fornece algo assim:

```
var lista = ["vermelho", "marrom", "azul", "preto", "amarelo"];

lista.capitalize();
```

Como o método `capitalize` não existe, receberemos o erro da figura 11.10.

```
> var lista = ["vermelho", "marrom", "azul", "preto", "amarelo"];
  lista.capitalize();
✖ ▶ TypeError: Object vermelho,marrom,azul,preto,amarelo has no method 'capitalize'
```

Figura 11.10: Não existe o método capitalize

Você pode fazer de forma *procedural*, criando uma função `capitalize` que recebe uma lista como parâmetro e retorna outra lista com a primeira letra de cada nome em maiúscula:

```
function capitalize(list) {
  var resultado = new Array();

  for(var pos = 0; pos < lista.length; pos++) {
    var primeiro = lista[pos].slice(0, 1);
    var resto = lista[pos].substr(1);
    resultado.push(primeiro.toUpperCase() + resto);
  }

  return resultado;
}
```

Agora podemos chamar a função `capitalize` passando a mesma lista como parâmetro. Funciona, mas não estamos usando orientação a objetos aqui.

```
var lista = ["vermelho", "marrom", "azul", "preto", "amarelo"];

capitalize(lista);
```

Podemos ver o resultado na figura 11.11.

```
> var lista = ["vermelho", "marrom", "azul", "preto", "amarelo"];

  capitalize(lista);
  ["Vermelho", "Marrom", "Azul", "Preto", "Amarelo"]
```

Figura 11.11: Usando uma função

Um problema que temos aqui é que a função `capitalize`, apesar de ter sido feita para trabalhar com `Arrays`, aceita qualquer coisa como parâmetro, podendo causar erros estranhos e que podem ser difíceis de encontrar.

Usando prototipação, podemos fazer com que um `Array` tenha um método chamado `capitalize`. Executando o mesmo código da figura 11.10, receberemos uma lista com a primeira letra de cada nome em maiúscula, em vez de um erro.

```
Array.prototype.capitalize = function() {
  var resultado = new Array();
```

```
  for(var pos = 0; pos < this.length; pos++) {
    var primeiro = this[pos].slice(0, 1);
    var resto = this[pos].substr(1);
    resultado.push(primeiro.toUpperCase() + resto);
  }

  return resultado;
}
```

Finalmente podemos executar o método `capitalize` para retornar a lista do jeito que queremos, como podemos ver na figura 11.12. Além de usarmos uma abordagem realmente orientada a objetos, podemos ter certeza de que sempre estaremos trabalhando em cima de uma lista, evitando erros causados por algum tipo de dado errado.

```
> var lista = ["vermelho", "marrom", "azul", "preto", "amarelo"];

  lista.capitalize();
  ["Vermelho", "Marrom", "Azul", "Preto", "Amarelo"]
```

Figura 11.12: Usando um método

Como cautela e caldo de galinha nunca fizeram mal a ninguém (apesar de ninguém ter pedido a opinião da galinha), não é demais lembrar que alterar qualquer objeto padrão da linguagem é tão perigoso quanto poderoso. Você pode facilmente quebrar completamente a sua aplicação alterando um objeto que é usado por bibliotecas de terceiros.

"Com grandes poderes vêm grandes responsabilidades"
– Benjamin Parker

11.4 Usando herança

Como agora você já sabe como criar objetos usando um construtor e já sabe como usar prototipação, por que não aprender também como simular herança clássica com JavaScript?

A figura 11.13 é um diagrama de classes.

Através dele podemos ver que todo Mamífero é um Animal, e todo Gato é um Mamífero. Por inferência, podemos dizer que todo Gato é um Animal, certo?

Isso significa que a classe Gato herda todas as características, propriedades e métodos das classes Mamífero e Animal. Chamamos isso de 'herança clássica'.

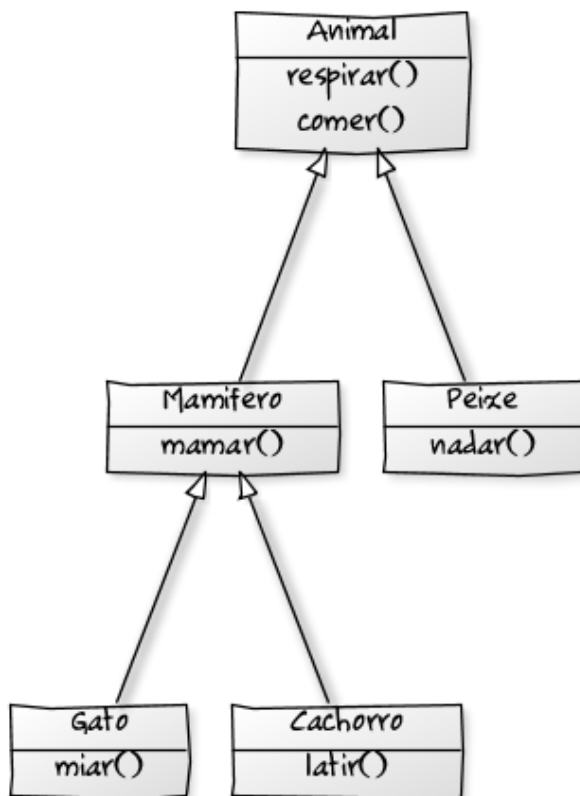
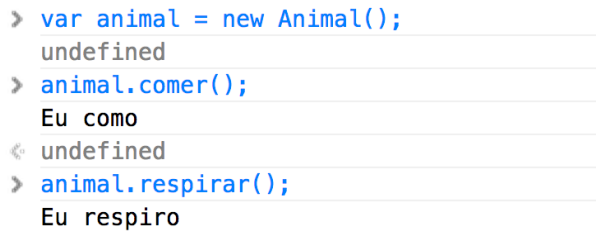



Figura 11.13: Um gato é um animal

Se, conforme já foi dito, JavaScript não tem classes, como podemos usar herança clássica?

Simples. Como foi explicado ao falarmos sobre prototipação, vamos herdar as características de outro objeto para simular a herança clássica.

Primeiro, vamos criar um construtor `Animal`. Um `animal` tem como característica as ações `comer` e `respirar`.

```
function Animal() {
```

```
  this.comer = function() {
    console.log("Eu como");
  };

  this.respirar = function() {
    console.log("Eu respiro");
  };
}

var animal = new Animal();
animal.respirar();
```

```
> var animal = new Animal();
  undefined
> animal.comer();
  Eu como
< undefined
> animal.respirar();
  Eu respiro
```

Figura 11.14: Implementando um Animal

Agora vamos criar um construtor `Mamifero`, que tem como característica a ação de `mamar`. A implementação é bem parecida com a função `Animal`:

```
function Mamifero() {
  this.mamar = function() {
    console.log("Eu mamo");
  }
}
```

Na figura 11.15 podemos ver que o método `mamar` é executado corretamente, mas nosso mamífero não come. Se ele é um animal, é esperado que ele coma.

```
> mamifero.mamar();
Eu mamo
< undefined
> mamifero.comer();
✖ ▶ TypeError: Object #<Mamifero> has no method 'comer'
```

Figura 11.15: Só que ele não come

Como podemos fazer para que o nosso `Mamifero` aja como um `Animal` sem que tenhamos que reescrever o código? A resposta está no protótipo.

Vamos usar o que aprendemos sobre protótipos para herdar as características de `Animal`:

```
Mamifero.prototype = new Animal();

mamifero = new Mamifero();

mamifero.mamar();

mamifero.comer();

mamifero.respirar();
```

Agora o nosso mamífero mama como qualquer `Mamifero`, come e respira como qualquer `Animal`. Podemos ver isso na prática pela figura 11.16.

```
> mamifero.mamar();
Eu mamo
< undefined
> mamifero.comer();
Eu como
< undefined
> mamifero.respirar();
Eu respiro
```

Figura 11.16: Agora o nosso Mamifero é um Animal

Vamos fazer o mesmo com `Gato` e `Cachorro` para comprovarmos se um `Gato` e um `Cachorro` continuam com as características de `Animal`. Vamos alterar o

comportamento do método `comer` do `Cachorro`. A isso chamamos de **sobrescrita de método** ou *method override*.

```
function Gato() {
  this.miar = function() {
    console.log("Miau!");
  }
}

Gato.prototype = new Mamifero();

function Cachorro() {
  this.latir = function() {
    console.log("Au au!");
  }

  this.comer = function() {
    console.log("Eu como ração");
  }
}

Cachorro.prototype = new Mamifero();

var bichano = new Gato();
var rex = new Cachorro();
```

Agora, na figura 11.17 vamos demonstrar que um `Gato` é um `Mamifero` e um `Animal`.

```
> bichano.respirar();
  Eu respiro
< undefined
> bichano.comer();
  Eu como
< undefined
> bichano.mamar();
  Eu mamo
< undefined
> bichano.miar();
  Miau!
```

Figura 11.17: Gato fazendo coisas que Gato faz

E na figura 11.18 demonstramos que, além de `Cachorro` ser `Mamifero` e `Animal`, também mudamos o comportamento do método `comer`.

```
> rex.respirar();
  Eu respiro
< undefined
> rex.mamar();
  Eu mamo
< undefined
> rex.latir();
  Au au!
< undefined
> rex.comer();
  Eu como ração
```

Figura 11.18: Cachorro ensinando override

11.5 Mixin no JavaScript

Mixin é uma forma de adicionar comportamentos e características a uma classe já existente, evitando que você tenha que reescrever o mesmo código várias vezes em lugares diferentes. Mixin também pode ser considerado como uma forma de se implementar herança múltipla, que é quando uma classe herda as características de mais de uma classe ao mesmo tempo. Na figura 11.19 podemos ver como isso funciona na prática.

Assim como na figura 11.13, que vimos na seção anterior, temos uma classe `Mamifero` que herda as características e comportamentos de `Animal`.

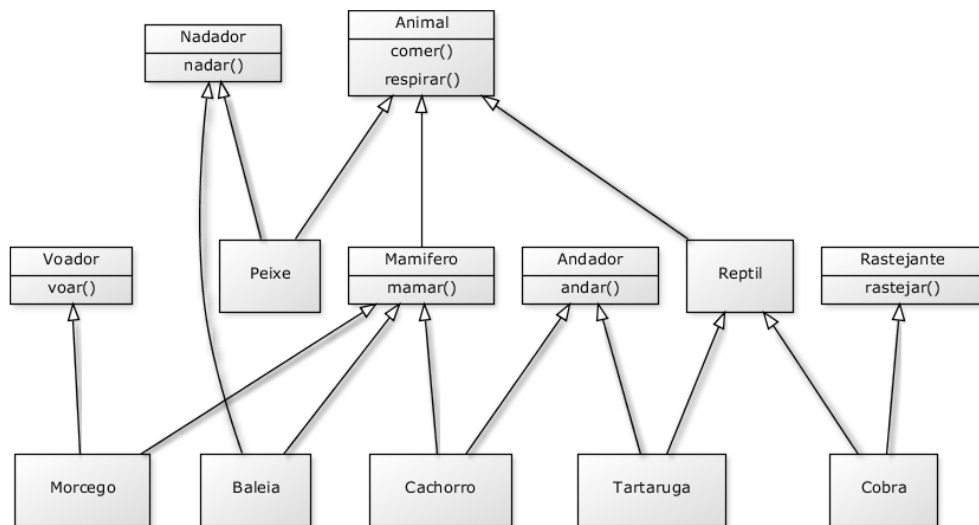



Figura 11.19: Herança múltipla é uma confusão só

Como você já deve ter imaginado, o JavaScript não implementa herança clássica e, com isso, também não dá suporte a herança múltipla.

O que você não deve ter imaginado é que, sendo o JavaScript uma linguagem tão poderosa e flexível, é possível que você mesmo a programe para permitir o uso de herança múltipla na forma de mixin.

Como todos os objetos do JavaScript herdam as características de `Object`, vamos criar um método `includes` no protótipo de `Object` para que você possa usá-lo com todos os objetos do JavaScript.

Vamos fazer com que esse método aceite um parâmetro chamado `constructor`, que representa o objeto cujos comportamentos e propriedades você quer herdar.

```
Object.prototype.includes = function (constructor) {

};
```

Como já sabemos, um objeto herda as características de outro objeto, então temos que criá-lo a partir do construtor que recebemos por parâmetro:

```
var objeto = new constructor();
```

Agora vem a parte que faz todo o trabalho pesado.

Vamos usar um `for` para pegar cada uma das propriedades que queremos herdar para o nosso objeto. Vamos usar o método `hasOwnProperty` para termos certeza de que a propriedade que vamos herdar vem mesmo do construtor que recebemos por parâmetro. Se o construtor herdou alguma característica, vamos ignorá-la. Você pode ignorar esse passo, se souber o que está fazendo.

Em seguida, vamos adicionar a propriedade ao protótipo do nosso próprio objeto. Lembra quando dissemos que objetos JavaScript são dicionários, e que dicionários podem ser acessados como Arrays, só que com qualquer tipo de valor como chave? Pois foi exatamente o que fizemos, passando o nome de `propriedade` como sendo a chave e copiando a propriedade a ser herdada para dentro do nosso objeto.

```
for (var propriedade in objeto) {
  if (objeto.hasOwnProperty(propriedade)) {
    this.prototype[propriedade] = objeto[propriedade];
  }
}
```

O nosso código vai ficar assim:

```
Object.prototype.includes = function (constructor) {
  var objeto = new constructor();

  for (var propriedade in objeto) {
    if (objeto.hasOwnProperty(propriedade)) {
      this.prototype[propriedade] = objeto[propriedade];
    }
  }
};
```

Vamos criar o nosso construtor `Animal`:

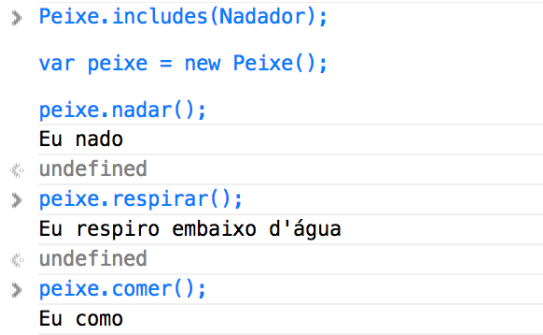
```
function Animal() {
  this.comer = function() {
    console.log("Eu como");
  };

  this.respirar = function() {
    console.log("Eu respiro");
  };
}
```

Criaremos um construtor para `Peixe`, lembrando que todo `Peixe` é um `Animal`. Vamos sobrescrever o método `respirar` e assim saberemos que tudo está funcionando conforme o previsto:

```
function Peixe() {
  this.respirar = function() {
    console.log("Eu respiro embaixo d'água");
  };
}

Peixe.prototype = new Animal();

var peixe = new Peixe();

peixe.nadar();
```

```
> var peixe = new Peixe();

  peixe.nadar();
✖ ▶ TypeError: Object #<Animal> has no method 'nadar'
```

Figura 11.20: Esse peixe ainda não nada

Queremos que o nosso `Peixe` nade, o que ainda não acontece, como podemos ver na figura 11.20. Podemos escrever um método `nadar` dentro de `Peixe`, e depois repetir o código no construtor `Baleia` (que ainda não escrevemos, mas está no diagrama de classes), que é `Mamifero`, ou podemos criar um construtor `Nadador`.

```
function Nadador() {
  this.nadar = function() {
    console.log("Eu nado");
  }
}
```

Note que `Nadador` é um construtor comum, sem nada de especial. Vamos usar aquele método `includes` para incluir as características de `Nadador` nos nossos objetos do tipo `Peixe`.

```
Peixe.includes(Nadador);
```

```
var peixe = new Peixe();

peixe.nadar();
```

Agora sim. Como podemos ver na figura 11.21, o nosso peixe respira como um `Animal`, e nada como um `Nadador`. Usando mixin, herdamos as características de dois objetos diferentes dentro de `Peixe`.

```
> Peixe.includes(Nadador);

  var peixe = new Peixe();

  peixe.nadar();
  Eu nado
< undefined
> peixe.respirar();
  Eu respiro embaixo d'água
< undefined
> peixe.comer();
  Eu como
```

Figura 11.21: Esse peixe ainda não nada

Agora que você já sabe criar objetos e trabalhar com encapsulamento, prototipação, herança e mixin, vamos aproveitar o próximo capítulo e aprender também a usar programação funcional com JavaScript.

Capítulo 12

Um pouco de programação funcional

"JavaScript é LISP disfarçado de C"
– Douglas Crockford

12.1 O que é programação funcional

Programação funcional é, assim como a orientação a objetos, uma forma de se pensar em como resolver problemas.

A base do que hoje é a programação funcional foi criada paralelamente por Alan Turing e Alonzo Church na década de 1930, antes mesmo da existência dos computadores como os conhecemos[12].

Infelizmente, a programação funcional passou muito tempo restrita aos meios acadêmicos, o que faz com que o iniciante no assunto fique assustado com toda aquela notação matemática e com os termos incompreensíveis.

$$e ::= x \mid \lambda x{:}\tau.e \mid e\ e$$

$$e ::= x \mid \texttt{(}x\texttt{) ->}\ \texttt{ifft(}\tau\texttt{,}x\texttt{);}\ e \mid e\ e$$

Figura 12.1: Notação matemática assusta

Para nossa sorte, o conceito de programação funcional é muito simples. Assim como na orientação a objetos a menor parte de um sistema é um objeto, sendo que você pode atribuir objetos a variáveis, pode passá-los por parâmetro e ter métodos retornando objetos, na programação funcional a menor parte do seu sistema é uma função.

Isso implica que você pode atribuir funções a variáveis, pode passá-las por parâmetro e mesmo fazer com que uma função retorne outra função. Algumas linguagens implementam também imutabilidade, que é quando todo valor é tratado como se fosse uma constante, e outros conceitos periféricos, mas não é o caso do JavaScript. Todos os demais conceitos de programação funcional derivam do fato de você lidar com funções como se fossem um valor como qualquer outro.

12.2 High order functions

Uma high order function (não achei uma tradução decente para o Português), apesar do nome intimidador, é basicamente uma função que recebe outra função como parâmetro ou devolve uma função como resultado. Quando você usa callbacks no JavaScript e no jQuery, você está fazendo uso de high order functions.

```
$("button.mallandro")
 .click(function() {
  alert("Ié ié!");
 });
```

O método `click` é uma high order function, e a função anônima que faz *Ié ié!* é um callback.

Sim, estamos apenas dando nomes a artifícios que já estamos utilizando há um bom tempo.

12.3 Escopo

Apesar do conceito de escopo não ser exclusivo da programação funcional, é importantíssimo que você entenda como funciona o escopo no JavaScript para que não fique confuso ao ver *closures* e *currying*.

Como na maioria das linguagens comerciais, uma variável declarada em um escopo maior é visível em um escopo menor, enquanto o contrário não é verdadeiro.

Na prática significa que uma variável global é visível por todo mundo:

```
var x = 1;

function variavelGlobal() {
  console.log(x);
}

variavelGlobal();
```

```
foo();
1
```

Figura 12.2: Valor de x não muda

Significa também que uma variável local só é vista dentro do escopo em que foi criada, ainda que tenha o mesmo nome de uma variável global:

```
var x = 1;

function variavelLocal() {
  var x = 99;
  var y = 42

  console.log(x, y);
}

variavelLocal();

console.log(x);

console.log(y);
```

```
bar();
99 42
undefined
> console.log(x);
1
undefined
> console.log(y);
▶ ReferenceError: y is not defined
```

Figura 12.3: Valor local de x

Quando uma função altera o valor de uma variável global, isso afeta toda a aplicação. Por isso o uso de variáveis globais não é considerado uma boa prática. Porém, uma variável global passada por parâmetro para uma função não tem o seu valor alterado:

```
var x = 1;
var y = 11;

function incremento(x) {
  console.log("Dentro: ", x, y);
  x++;
  y++;
  console.log("Dentro: ", x, y);
}

incremento(x);
// Dentro:  1 11
// Dentro:  2 12

console.log("Fora: ", x, y);
// Fora:  1 12
```

```
> meh(x);
  Dentro:  1 11
  Dentro:  2 12
< undefined
> console.log("Fora: ", x, y);
  Fora:  1 12
< undefined
```

Figura 12.4: Variável global sendo alterada

12.4 Closures

Outra característica do escopo é que uma função guarda as variáveis do contexto em que foi criada. Isso significa que uma função pode continuar acessando variáveis que só existiam no momento em que ela foi criada.

```
function counter() {
  var x = 0;

  return function() {
    return ++x;
  }
}

var count = counter();

console.log(count());

console.log(count());

console.log(count());

console.log(count());

console.log(x);
```

```
> console.log(count());
1
< undefined
> console.log(count());
2
< undefined
> console.log(count());
3
< undefined
> console.log(count());
4
< undefined
> console.log(x);
✖ ▶ ReferenceError: x is not defined
```

Figura 12.5: Contador usando closure

O que acontece aqui é que `count` recebe uma função que incrementa o valor da variável `x`, só que essa variável existe apenas dentro da função `counter`. O que ocorre é que a função armazenada em `count` *se lembra* da variável que foi criada em outra função, mas que não está mais sendo executada.

Ao tentarmos exibir o valor da variável `x`, recebemos um erro, pois ela não existe no escopo global, como podemos ver na imagem 12.5.

12.5 Currying

Juntando tudo o que vimos até aqui sobre high order functions, escopo e closures, chegamos ao *currying*. Currying é uma operação em que você transforma uma função que receberia mais de um parâmetro em uma série de chamadas de funções com apenas um parâmetro cada.

Um dos usos dessa técnica é evitar, de forma elegante, que o mesmo parâmetro seja passado para a mesma função.

Vamos pegar um exemplo escrito de forma imperativa. Temos uma função `hey`, que recebe os parâmetros `texto` e `nome` e, a partir disso, imprime uma saudação.

```
function hey(texto, nome) {
  console.log(texto + ", " + nome);
}
```

```
hey("Bom dia", "João");

hey("Bom dia", "José");

hey("Bom dia", "Nicolau");
```

```
> hey("Bom dia", "João");
  Bom dia, João
< undefined
> hey("Bom dia", "José");
  Bom dia, José
< undefined
> hey("Bom dia", "Nicolau");
  Bom dia, Nicolau
< undefined
```

Figura 12.6: Bom dia imperativo

Você pode dizer que poderíamos guardar *Bom dia* em uma variável. Concordo, mas isso não mudaria nada dentro do que estamos apresentando.

Em algumas linguagens, currying é algo natural e pode ser feito de forma transparente. No JavaScript precisamos fazer manualmente, mas isso torna a coisa toda mais fácil de ser compreendida.

Reescrevendo a mesma função usando currying, teremos o código abaixo:

```
function hey(texto) {
  return function(nome) {
    console.log(texto + ", " + nome);
  }
}

var bomDia = hey("Bom dia");

bomDia("João");

bomDia("José");

bomDia("Nicolau");
```

```
> bomDia("João");
  Bom dia, João
<· undefined
> bomDia("José");
  Bom dia, José
<· undefined
> bomDia("Nicolau");
  Bom dia, Nicolau
```

Figura 12.7: Bom dia funcional

Programação funcional é muito mais do que os conceitos que apresentei aqui, mas como um primeiro contato já dá para fazer muita coisa bacana.

Recomendo que você estude e pesquise a respeito. Mesmo que você não use uma linguagem funcional no seu dia-a-dia, o fato de conhecer um novo modo de pensar acaba alterando a forma como você resolve problemas e fazendo com que você tenha ideias melhores e mais elegantes.

Há linguagens que vão levar alguns desses conceitos para dentro dela mesma, tornando currying, closure e outros artifícios completamente transparente.

Agora que você já domina orientação a objetos e programação funcional no JavaScript, criar seus próprios plugins no jQuery vai parecer brincadeira de criança. E é exatamente isso que vamos fazer no próximo capítulo.

Capítulo 13

Criando plugins para jQuery

"Por favor, não tente deixar o seu código ilegível. Eu garanto que ele ficará ilegível o bastante sem que você se esforce."
– "Cowboy" Ben Alman

13.1 O que são plugins?

O jQuery não foi desenvolvido para resolver todos os problemas, imaginados ou não, que um desenvolvedor pode encontrar ao criar uma aplicação para Web.

Ao invés de desperdiçar tempo e esforço nessa tarefa impossível (o que alguns frameworks famosos continuam tentando fazer), John Resig e os desenvolvedores do jQuery resolveram criar um núcleo bem definido e facilmente extensível.

Com isso, qualquer um com um pouco mais de curiosidade pode utilizar a base que o jQuery fornece para criar métodos que resolvam seus problemas. Esses métodos são integrados de tal forma ao jQuery que passam a ser tratados como se fossem parte da biblioteca. Cada conjunto de métodos desses é chamado de plugin.

Um dos motivos para o jQuery ser tão popular é a imensa quantidade de plugins desenvolvidos pela comunidade e empresas especializadas. Existem plugins para quase tudo o que você imaginar e, se não existir, você pode muito bem criar e compartilhar.

Para todos os códigos e exemplos que vamos apresentar aqui, é necessário que você inclua o jQuery no seu HTML ou no jsFiddle.

13.2 A anatomia de um plugin

Um plugin é um objeto que você agrega ao objeto `jQuery`, e que deve seguir algumas boas práticas para manter uma boa relação com os outros milhares de plugins existentes.

Algumas dessas boas práticas são tidas como padrão pelos desenvolvedores e usuários mais experientes. Outras são apenas questão de bom senso e surgem depois de você cansar de ter plugins dando problema em produção e entrando em conflito com outros plugins ou bibliotecas.

A primeira coisa que você deve saber sobre um plugin jQuery é que todas as chamadas, consultas e configurações de um plugin devem ser feitas através de um único método.

Usando como exemplo um dos primeiros plugins que desenvolvi, vou demonstrar a maneira errada de usar e que acaba sendo feita por muitos incautos desenvolvedores.

O plugin, que vamos chamar de `jeitoErrado`, tem uma opção para ser aplicado e outra opção para ser removido.

O jeito errado consiste em criar um método para cada operação. No caso, um método para aplicar e outro para remover.

```
$(".qualquer").jeitoErrado();

$(".qualquer").removerJeitoErrado();
```

Segundo a documentação do jQuery[10], essa forma enche o objeto `jQuery` de sujeira, aumentando as chances de seu plugin afetar o funcionamento de outros plugins.

O correto seria passar o comando de remoção por parâmetro para o próprio método `jeitoErrado`.

```
$(".qualquer").jeitoCerto();

$(".qualquer").jeitoCerto("remove");
```

Outra boa prática é permitir que o plugin funcione sem que haja a necessidade de se passar parâmetros no método que o aplica. Obviamente ele pode não funcionar como você gostaria, mas é sempre uma boa prática definir valores padrão para cada parâmetro, para o que o plugin funcione sempre.

```
$(".bonito").textoBonito();

$(".bonito").textoBonito({corDaLetra: "#00ff00"});

$(".bonito").textoBonito({corDaLetra: "#00ff00", corDeFundo: "#ff00ff"});
```

Apesar de não ser sempre possível, é uma boa prática permitir que você altere parâmetros depois que o plugin já foi aplicado. Para isso, devemos lembrar de usar sempre o mesmo método.

```
$(".bonito").textoBonito();

$(".bonito").textoBonito("corDeFundo", "#ff00ff");
```

Sempre que possível, faça com que o usuário possa consultar o valor atual de alguma das configurações. Para isso, use o próprio nome da configuração como um parâmetro.

```
$(".bonito").textoBonito();

$(".bonito").textoBonito("corDeFundo", "#ff00ff");

alert($(".bonito").textoBonito("corDeFundo"));
```

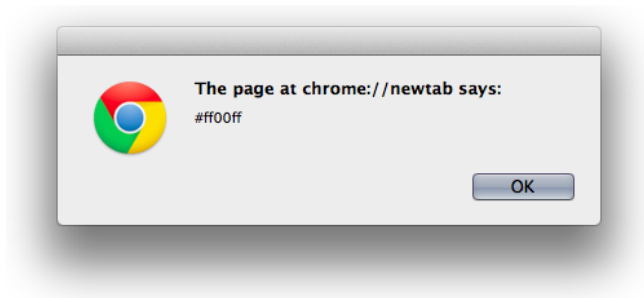



Figura 13.1: O valor de corDeFundo

Uma das coisas mais bacanas no jQuery, como já mostramos, é o fato de você poder selecionar os elementos apenas uma vez e executar quantas operações forem necessárias em cima deles. Além do seu código ficar mais limpo, você ganha em performance, pois só vai gastar tempo selecionando os elementos uma vez.

Por conta disso, nunca se esqueça de fazer com que o seu plugin seja *encadeável* (interface fluente), fazendo o seu método retornar o objeto `jQuery` que você alterou.

Ainda usando aquele meu plugin errado como exemplo, veja como ele foi feito. Se eu tentasse encadear qualquer chamada após o `pluginErrado`, eu receberia um erro. Isso me obrigava a selecionar os elementos mais de uma vez.

Apenas a título de demonstração, a partir do jQuery 1.9 você pode enviar um objeto para o método `css` em vez de fazer várias chamadas seguidas.

```
$(".qualquer").pluginErrado();

$(".qualquer")
  .pluginQualquer()
  .css({"color": "red", "text-shadow": "0.5em #ccc"});
```

O jeito correto, mantendo a política da boa vizinhança, é manter o meu plugin encadeável e *não quebrar a corrente*.

```
$(".qualquer")
  .bomVizinho()
  .pluginQualquer()
  .css({"color": "red", "text-shadow": "0.5em #ccc"});
```

Use nomes em inglês

Apesar de eu ter usado nomes de métodos e propriedades em português, com o único objetivo de apresentar claramente as boas práticas, eu recomendo fortemente que você sempre use nomes em inglês.

Deixando de lado toda a discussão sobre soberania nacional e imperialismo, tente imaginar que você encontrou um plugin muito bacana com nomes em chinês ou tailandês. Quais as chances de você utilizá-lo?

Gostando ou não, o inglês é o idioma oficial da Internet e do desenvolvimento de softwares.

Por fim, lembre-se de que você não está sozinho no mundo e que o seu plugin é, por enquanto, apenas um tijolinho insignificante numa muralha de opções e plugins disponíveis.

Agora vou mostrar como criar nosso próprio plugin usando essas boas práticas.

13.3 Escrevendo a declaração do plugin

Como já dissemos, um plugin é invocado através de um método que vamos adicionar no objeto `jQuery`. A diferença aqui é que, ao invés de adicionar o método em `prototype`, vamos adicionar em `fn`.

Não se preocupe, pois não há novidade nenhuma aqui. Esse `fn` é apenas um apelido para `prototype`, conforme podemos ver em um trecho do código fonte do jQuery:

```
jQuery.fn = jQuery.prototype = {
  // e mais uma tonelada de código
```

Vamos criar um plugin que verifica se o CPF informado está correto e, no auge da criatividade, vamos chamar de `validador`. Permitiremos também que o usuário utilize as próprias validações.

Figura 13.2: Como nosso validador vai ficar

Para criar um plugin, começamos criando um método chamado `validador`, que adicionaremos ao protótipo `fn` do objeto jQuery. Como será possível ao usuário informar uma função personalizada caso ele queira usar outro tipo de validação, vamos receber as opções como parâmetro:

```
jQuery.fn.validador = function(options) {
  // esse é o menor plugin que existe
}
```

Mas perceba que estamos usando `jQuery` o invés de usarmos a forma reduzida `$`. O problema de usar `$` é que outras bibliotecas também usam esse símbolo como uma forma reduzida. A biblioteca *Prototype* é uma que faz uso desse símbolo o tempo todo.

Para usarmos o símbolo `$` sem medo de causar conflito com outras bibliotecas, vamos usar uma técnica chamada **IIFE**, que em tradução livre significa *Invocação imediata de função* ou, numa explicação bem mais simples, *uma função que invoca a si mesma*.

Apesar do nome pomposo, *IIFE* significa que vamos definir uma variável `$` dentro do escopo de uma função para que possamos usá-la sem que haja interferência de qualquer outra biblioteca. Vamos passar `jQuery` como parâmetro para a variável `$` e não precisaremos nos preocupar mais com isso.

Na prática, vamos declarar a função entre parênteses, passando o objeto `jQuery` como parâmetro:

```
(function($) {
  // e aqui vem todo o código do nosso plugin
})(jQuery);
```

Apesar de parecer confuso e pouco legível à primeira vista, esse código faz exatamente o mesmo que o trecho abaixo, mas sem criar nenhuma variável:

```
var escopo = function($) {
  // e aqui vem todo o código do nosso plugin
}

escopo(jQuery);
```

Quando você deixa de criar uma variável para guardar a função, você reduz as chances de causar conflito com código de terceiros. Imagine que algum outro plugin definiu uma variável no escopo global com o mesmo nome que a sua.

Uma outra boa regra diz que devemos inserir um ponto-e-vírgula *antes* do código no nosso plugin. Como as regras que definem o uso opcional de ponto-e-vírgula[16] no final de uma instrução não são inteiramente compreendidas pela maioria dos desenvolvedores, é seguro encerrarmos qualquer instrução que esteja aberta com um ponto-e-vírgula no começo do nosso código.

Mesmo se o código de terceiros que estiver imediatamente antes do nosso tiver sido fechado corretamente, o nosso ponto-e-vírgula a mais não vai fazer a menor diferença.

Lembre-se de que não fazemos a menor ideia do ambiente e das condições em que nosso plugin será usado.

Com isso podemos alterar o código do nosso plugin para usar o símbolo `$`:

```
;(function($) {
  $.fn.validador = function(options) {
    // esse é o segundo menor plugin que existe
  }
})(jQuery);
```

Vamos criar uma caixa de texto com HTML e fazer com que o nosso plugin seja aplicado a ele.

A declaração HTML não tem segredo nenhum. O elemento terá um `id` com o valor `cpf`, que usaremos na hora de aplicar o plugin.

```
<p>
  <label for="cpf">CPF: </label>
  <input type="text" id="cpf" maxlength="14" size="18">
</p>
```

E após o código do nosso plugin, vamos fazer o jQuery aplicá-lo ao elemento assim que o documento for carregado

```
$(function() {
  $("#cpf").validador();
})
```

Agora vamos testar e nada acontece. Como não escrevemos código para o plugin, ele foi aplicado corretamente, mas ainda não faz nada.

13.4 O algoritmo de CPF

Apesar de não fazer parte do assunto principal deste livro, vou explicar rapidamente como funciona o algoritmo de validação de CPF para que você possa acompanhar o código deste capítulo.

Vamos criar uma função chamada `validaCpf`, que vai receber um texto e retornar verdadeiro somente se o CPF for válido.

Vamos deixar essa função dentro do nosso escopo, para que não haja o risco de haver outra função com o mesmo nome, criada por terceiros.

```
;(function($) {

  var validaCpf = function(text) {

  }

  // código do plugin

})(jQuery);
```

O nosso texto pode ter qualquer caractere como separador. O usuário pode usar ponto, traço, barra ou qualquer outra coisa que passar pela cabeça. Em vez de imaginarmos todas as possibilidades, vamos arrancar do texto tudo que não seja um dígito. Para isso, vamos usar o método `replace` com um recurso chamado **expressões regulares**.

Expressões regulares são assunto suficiente para um livro inteiro. O que precisamos saber no JavaScript é que uma expressão regular é delimitada por `/ /`, assim como um texto é delimitado por aspas. Existe um código `\D` para selecionar um caractere que não é um dígito e, por fim, vamos passar o parâmetro `g` após a última barra para dizer que a alteração é *global*, ou seja, deve ser aplicada em todo o texto.

Uma vez extraídos os dígitos, vamos verificar se o texto tem exatamente 11 dígitos. Caso contrário, retornaremos `false` sem verificar mais nada.

```
text = text.replace(/\D/g, "");

if(text.length === 11) {
  // aqui vem o algoritmo de verificação
}

return false;
```

Vamos converter o texto para um `Array` de dígitos. Uma forma rápida de converter um dígito em formato texto para número é subtrair o ASCII do caractere por 48, que é o código do dígito `0`.

```
var digitos = [];

for(var pos = 0; pos < text.length; pos++) {
  digitos[pos] = text.charCodeAt(pos) - 48;
}
```

No primeiro passo, vamos calcular o primeiro dígito verificador. Para isso vamos multiplicar os nove primeiros dígitos por 10, 9, 8, e assim por diante respectivamente, somar os resultados da multiplicação e calcular o resto de divisão por 11 para obter o primeiro dígito. Se o resto da divisão for 10, vamos considerar o dígito como 0. Se o primeiro dígito não estiver de acordo, encerramos o algoritmo retornando `false`.

```
var soma = 0;

for(var pos = 0; pos < 9; pos++) {
  soma += digitos[pos] * (10 - pos);
}

var primeiroDigito = 11 - (soma % 11);

if(primeiroDigito > 9) {
```

```
  primeiroDigito = 0;
}

if(digitos[9] !== primeiroDigito) {
  return false;
}
```

Para calcularmos o segundo dígito verificador, multiplicaremos os dez primeiros dígitos do CPF por 11, 10, 9, e assim por diante respectivamente, somamos os valores obtidos e calculamos o resto da divisão por 11. Se o resto da divisão for 10, vamos considerar o dígito como 0.

Finalmente, retornamos o valor da função como sendo a comparação do número obtido com o segundo dígito verificador.

```
// continuação do código

  soma = 0;

  for(var pos = 0; pos < 10; pos++) {
    soma += digitos[pos] * (11 - pos);
  }

  var segundoDigito = 11 - (soma % 11);

  if(segundoDigito > 9) {
    segundoDigito = 0;
  }

  return digitos[10] === segundoDigito;
}
```

13.5 Adicionando funcionalidade ao plugin

Para fins de manter os exemplos simples, considere que o código que apresentarmos a partir daqui estará dentro do método `validador`. Ao final do capítulo, nós mostraremos o código completo para esclarecer qualquer dúvida que apareça.

A primeira coisa que precisamos prestar atenção é que não sabemos se o jQuery vai selecionar um ou mais elementos para o nosso plugin. Vamos considerar sempre o pior caso, que é quando o jQuery nos retorna uma lista de elementos. Mesmo

que ele nos retorne um ou nenhum elemento, o código a seguir vai continuar funcionando. Vamos usar o nosso já conhecido `each` para modificar cada um dos elementos.

O objeto jQuery `this` referencia os elementos que foram selecionados para que o plugin seja aplicado.

Para fazermos com que o nosso plugin não quebre a corrente, retorne o valor do método `each`. Esse método retorna o objeto que está sendo iterado, deixando tudo pronto para que o usuário possa encadear mais métodos futuramente.

Vamos criar uma função `init`, que configura o plugin e adiciona os eventos necessários aos elementos selecionados. Com isso, o método `each` apenas recebe `init` como um callback.

```
var init = function() {
}

return this.each(init);
```

Na função `init`, vamos criar uma variável `element` para não termos que ficar acessando `$(this)` todas as vezes que precisarmos. Essa variável vai conter a referência ao objeto ao qual o plugin está sendo aplicado.

Vamos também definir os parâmetros do plugin, usando o método `extend` do jQuery. O método `extend` recebe dois parâmetros: o primeiro é um objeto contendo as propriedades do plugin e seus valores padrão, enquanto o segundo parâmetro contém um objeto com as opções passadas pelo usuário. O resultado é um objeto contendo todas as opções do objeto mesclada com os valores alterados pelo usuário.

No caso do nosso plugin, teremos apenas um parâmetro `function`, que vai indicar a função a ser executada para validar o conteúdo da caixa de texto. Por padrão, o plugin verifica a validade do CPF informado.

As configurações do objeto estarão disponíveis na variável `settings`.

```
var element = $(this);
var settings = $.extend({
  "function" :  validaCpf
}, options);
```

Apenas para melhorar a legibilidade do código, vamos criar uma função chamada `isValid`, que executa a função indicada no parâmetro `function`.

```
var isValid = function(text) {
  return settings["function"](text);
};
```

Vamos criar também um callback que será executado quando o elemento perder o foco. Na prática, significa que, quando o elemento perder o foco, a validação será feita e a classe CSS `erro` será aplicada caso o valor seja inválido, ou `valido` quando o valor estiver de acordo com a validação.

```
var onElementBlur = function() {
  var text = element.val();

  if(isValid(text)) {
    element.removeClass("erro");
    element.addClass("valido");
  } else {
    element.removeClass("valido");
    element.addClass("erro");
  }
}
```

Por fim, encerramos o código do plugin atribuindo o callback `onElementBlur` ao evento `blur`. Note que estamos usando um namespace `validador`, para que isso não cause conflito com nenhum outro plugin.

```
element.on("blur.validador", onElementBlur);
```

Na figura 13.3 podemos ver como ficou o nosso plugin. Os arquivos CSS e imagens adicionais estão disponíveis no repositório do GitHub, cujo link eu apresentarei mais ao final do capítulo. O CPF usado como teste foi criado aleatoriamente por um gerador de CPF.

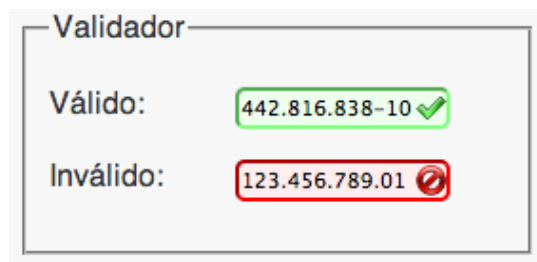



Figura 13.3: Um CPF válido e um inválido

13.6 Personalizando o plugin

Como você se lembra, nós criamos a possibilidade de usar validações personalizadas no nosso plugin.

Para demonstrar como isso funciona, vamos criar uma função muito simples, que apenas verifica se o campo está vazio ou não. Caso ele esteja vazio, faremos com que o plugin reclame e acuse um erro.

Vamos criar uma função chamada `obrigatorio`, que será passada em uma das opções do plugin.

```
function obrigatorio(text) {
  return text.trim() !== "";
}
```

E, para usarmos no plugin, basta declararmos o parâmetro `function`, selecionando uma caixa de texto com a classe igual a `obrigatorio`. O `id` do elemento e o nome da função ficam ao seu critério.

```
$(".obrigatorio").validador({function: obrigatorio});
```

Para testar, eu criei duas caixas de texto e obtive a saída apresentada na figura 13.4.

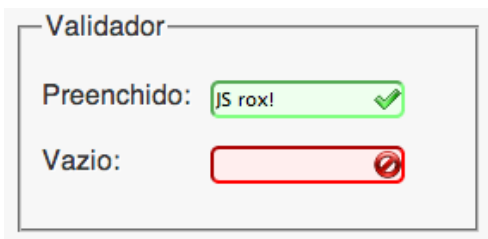



Figura 13.4: Campo obrigatório

13.7 Mais plugins

O site http://plugins.jquery.com é o repositório oficial de plugins do jQuery. Lá você encontra desde plugins muito bem escritos até alguns que poderiam ter sido tratados com maior cuidado.

A título de exemplo e referência, vou indicar alguns plugins que eu utilizo e recomendo.

Máscara

Nosso colega Diego Plentz reescreveu recentemente o plugin *MaskMoney*, com o qual tive o prazer de contribuir. É basicamente um plugin que possibilita a formatação de campos de texto para permitir a digitação de valores monetários.

Mexendo um pouco nas configurações, você consegue usar o plugin para qualquer tipo de número, inteiro ou decimal.

Para usarmos o plugin, você deve baixar o arquivo no repositório do GitHub e, considerando que você criou uma caixa de texto com o `id` igual a `valor`, o código JavaScript para lidar com valores em Real é:

```
$("#valor").maskMoney({symbol:     "R$",
                       thousands: ".",
                       decimal:   ","});
```

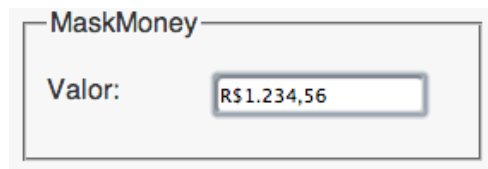



Figura 13.5: MaskMoney em funcionamento

O repositório do GitHub fica no endereço abaixo:

https://github.com/plentz/jquery-maskmoney

Gráficos

O jqPlot é uma ferramenta de geração de gráficos que utiliza o componente Canvas do HTML5.

Existem centenas de possibilidades diferentes de gráficos e temas. Vou apresentar um gráfico bem simples, onde você precisa informar as coordenadas X e Y em um `Array`:

```
$.jqplot('chartdiv',
  [[[1, 2],
```

```
[3, 5.12],
[5, 13.1],
[7, 33.6],
[9, 85.9],
[11, 219.9]]]);
```

O resultado pode ser visto na figura 13.6.

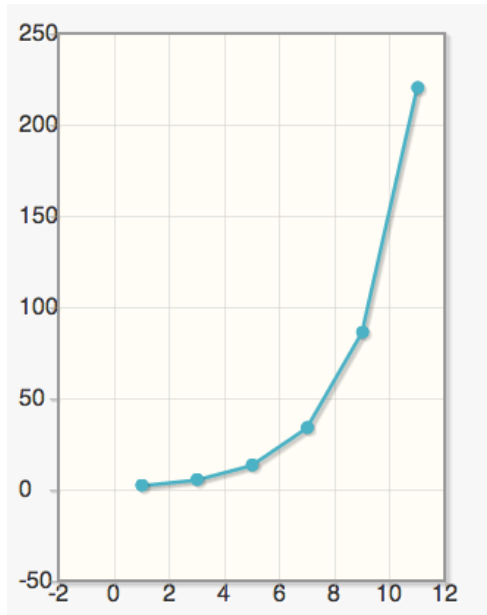



Figura 13.6: Um gráfico simples com jqPlot

Você pode fazer o download e consultar a documentação no endereço abaixo: http://www.jqplot.com/

13.8 Onde aprender mais

O código completo para o plugin está no capítulo 13 do repositório do livro no GitHub, no endereço abaixo:

https://github.com/pbalduino/livro-js-jquery

Existem mais possibilidades para desenvolver seu próprio plugin. Recomendo que você leia a documentação oficial no endereço abaixo e, como diversão, escreva plugins para ajudar a fixar o que aprender de novo.

http://docs.jquery.com/Plugins/Authoring

Disponibilizei um exemplo deste plugin, com a validação de CPF e a checagem de campo obrigatório no endereço abaixo:

http://jsfiddle.net/pbalduino/LLret/

Capítulo 14

Dicas para usar melhor o jQuery

"There is more to life than simply increasing its speed."
– Mahatma Gandhi

14.1 Por que performance é importante?

Além de ter um design impecável, é importantíssimo que a sua aplicação não demore para carregar ou executar as tarefas. O usuário associa diretamente a responsividade e a velocidade aparente de um site à qualidade do produto e à competência da empresa.

É sabido que o Google pode alterar a relevância do seu site de acordo com a performance. Sites mais lentos são considerados menos relevantes.

Recomendo a leitura de um excelente post do Sérgio Lopes sobre carregamento assíncrono de recursos no endereço http://goo.gl/fu88T, para que você melhore a responsividade do seu site em todos os aspectos.

Sabendo disso, é necessário que tenhamos em mente como fazer para que a nossa aplicação execute no menor tempo possível, usando a ferramenta do melhor modo.

Para não ficarmos no campo das especulações, vamos usar a ferramenta **jsPerf** para executar as avaliações de tempo. Você pode criar seus próprios testes em http://jsperf.com/

14.2 Use sempre a versão mais recente

Em cada nova versão do jQuery, informada pelos dois primeiros números, são incluídas melhorias de performance e limpeza de código. Suporte a browsers antigos são removidos, parte do código é reescrita de forma que execute mais rapidamente.

As versões menores, indicadas pelo terceiro número, contém correções de bugs e falhas de segurança.

Quando você utiliza a versão mais recente do jQuery, você faz uso de todas as melhorias de desempenho e segurança em relação a versões mais antigas.

Na figura 14.1 podemos ver a comparação de desempenho entre as versões 1.4.2, 1.8.0, 1.9.0 e 1.9.1, que é a mais recente enquanto escrevo este livro.

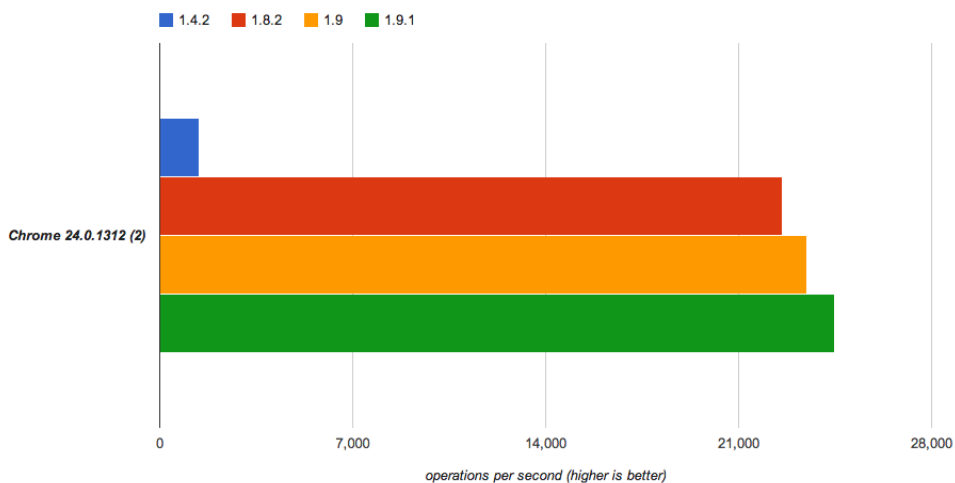



Figura 14.1: Versão - Operações por segundo (mais é melhor)

Opte sempre pela última versão do jQuery. A chance de seus scripts quebrarem costuma ser mínima se você adotar essa prática.

14.3 Escolha os seletores corretos para a tarefa

Sempre que você seleciona um ou mais elementos, o jQuery percorre a árvore do DOM até encontrar o que você procura.

Alguns seletores usam funções nativas do browser, como é o caso do `getElementById`, e é muito rápido. Outros dependem do quão novo é o seu browser, podendo variar entre o *rápido* e o *ridiculamente lento*, como acontece com o método `getElementsByClass`. Se você estiver usando uma versão desatualizada do Internet Explorer, uma busca por classe se mostrará lenta e pesada.

Outros seletores são lentos de qualquer maneira, como é o caso dos **pseudo-seletores**, que selecionam as linhas pares ou ímpares de uma tabela, por exemplo.

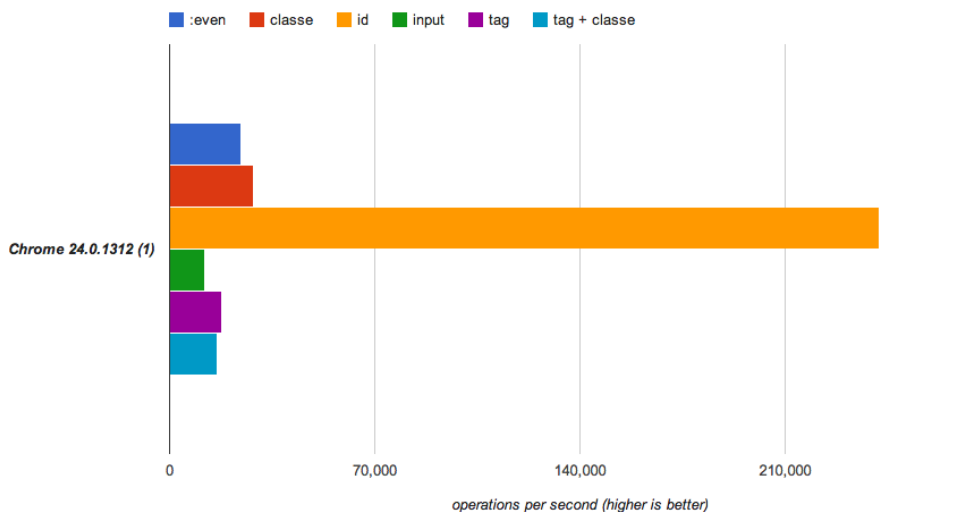



Figura 14.2: Seletores - Operações por segundo (Mais é melhor)

Na figura 14.2 temos a confirmação de que selecionar um elemento através de seu `id` é muito mais rápido.

Quando queremos selecionar mais de um elemento, o seletor de classes é uma boa escolha, se você estiver usando um browser moderno.

O pior caso de todos é quando selecionamos elementos de um formulário usando o tipo, como no código abaixo:

```
$("input[type='text']");
```

Sempre que possível, utilize classes ao invés de selecionar elementos através de alguma propriedade.

14.4 Não se esqueça do cache

Ainda mais efetivo do que escolher os seletores corretos é entender como usar o **cache de objetos**.

Fazer cache, no contexto do jQuery, nada mais é do que armazenar o resultado dos seletores em uma variável.

Ao invés de perder tempo selecionando várias vezes o mesmo conjunto de elementos, você pode muito bem selecionar uma vez, guardar o resultado numa variável para, em seguida, efetuar todas as operações necessárias.

A figura 14.3 mostra a diferença de desempenho entre diferentes abordagens.

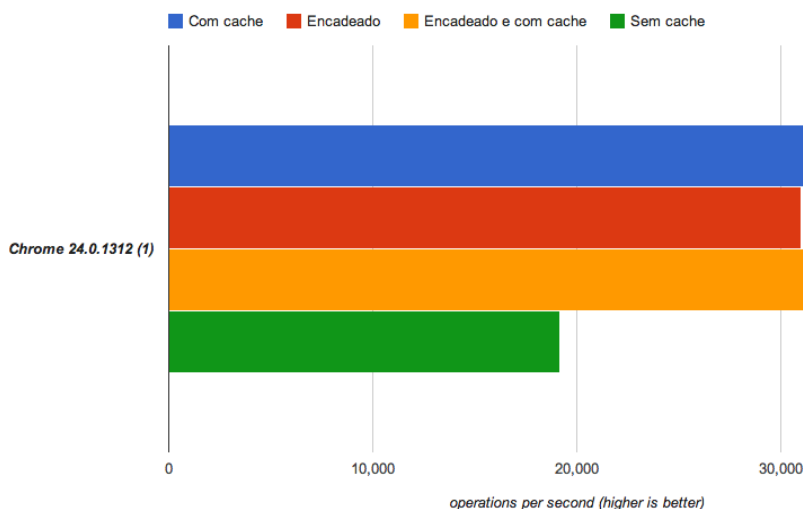



Figura 14.3: Cache - Operações por segundo (mais é melhor)

Para você entender o que foi feito, vamos usar um código HTML bem simples, contendo um elemento `div`:

```
<div id="teste"></div>
```

Escreveremos um código que efetua uma série de operações sobre esse elemento. Apenas a título de explicação, vamos fazê-lo desaparecer e aparecer algumas vezes.

Ignore o fato de que poderíamos usar um `for`. A ideia aqui é apenas apresentar uma boa prática.

```
$("#teste").hide();
$("#teste").show();

$("#teste").hide();
$("#teste").show();

$("#teste").hide();
$("#teste").show();
```

O que acontece aqui, internamente, é que o jQuery vai criar um objeto, parsear o seletor que foi passado, usar as funções nativas do browser para localizar o elemento que você está procurando através do `id` e só então retornar o objeto que contém o elemento que buscamos.

Depois, usando o objeto jQuery, será executada a operação desejada. No nosso caso, os métodos `hide` ou `show`.

Isso vai acontecer seis vezes. E olhe que aqui eu não entrei na questão de ter que alocar memória a cada vez que o objeto é criado e liberá-la assim que o processamento termina.

Com cache, nosso código vai selecionar o elemento apenas uma vez, com todos os procedimentos a que tem direito: criar um objeto jQuery, parsear o seletor, usar as funções nativas do browser, retornar o objeto com o elemento dentro e deixá-la confortavelmente numa variável.

Então, nas mesmas seis vezes, vamos apenas acessar a variável e executar as operações desejadas.

```
$cache = $("#teste")

$cache.hide();
$cache.show();

$cache.hide();
$cache.show();

$cache.hide();
$cache.show();
```

Essa segunda abordagem chega a ser duas vezes mais rápida do que a primeira, conforme apresentamos na figura 14.3.

Como já demonstramos várias vezes, o jQuery nos permite encadear as chamadas de métodos, fazendo com que usemos o seletor apenas uma vez.

Existem duas formas de se usar encadeamento. A primeira é encadear diretamente no seletor:

```
$("#teste")
  .hide()
  .show()
  .hide()
  .show()
  .hide()
  .show();
```

E a segunda forma é fazer um cache desse objeto e encadear os métodos na variável.

```
$cache = $("#teste");
$cache.hide()
  .show()
  .hide()
  .show()
  .hide()
  .show();
```

Como é obrigação de um desenvolvedor de softwares ser curioso, eu quis saber qual das duas versões oferece a melhor performance. A conclusão é que tanto faz.

O peso do que estamos fazendo está no processo de criar objeto, parsear e procurar. Uma vez que você otimize o seu código fazendo com que esse processo seja executado apenas uma vez, tanto faz se o resultado vai estar numa variável ou diretamente no retorno do seletor.

Você pode ver esse teste com detalhes no endereço abaixo:

http://jsperf.com/good-jquery-code-cache-dom-selectors/2

14.5 Às vezes, menos é mais

Como vimos até aqui, o jQuery resolve quase todos os nossos problemas de incompatibilidades entre browsers e simplifica muito a manipulação de DOM e a comunicação de dados usando AJAX.

Porém, como você deve ter cansado de ouvir, não existe bala de prata. Você não pode achar que o jQuery é a solução para todos os seus problemas. Em algumas vezes, a solução mais rápida é também a mais simples.

Nos casos em que pode ocorrer incompatibilidade entre browsers, use o jQuery para fugir de todos os problemas que vimos nos capítulos 3 e 4.

HTML

Em casos em que você precise apenas ler ou modificar o conteúdo de um elemento que você acessa através do `id`, considere a possibilidade de usar JavaScript puro ao invés de delegar a tarefa ao jQuery.

A figura 14.4 mostra o resultado do teste comparando o uso de jQuery e JavaScript puro para alterar e ler o conteúdo de um elemento.

Considerando que você tenha um elemento `div` qualquer com `id` igual a `teste`, usamos as instruções abaixo para os testes de gravação:

```
document.getElementById("teste").innerHTML = "texto";

$("#teste").html("texto");
```

A primeira instrução foi quase duas vezes e meia mais rápida do que a primeira. Se levarmos em conta que praticamente todos os browsers em uso atualmente tem o método `getElementById` implementado, temos ai um ganho considerável.

Para o teste de leitura, usamos o código abaixo:

```
document.getElementById("teste").innerHTML;

$("#teste").html();
```

Como era de se esperar, a leitura é muito mais rápida que a gravação. O que impressionou foi a diferença de performance entre as versões: o código que usa JavaScript puro foi quase onze vezes mais rápido que a versão jQuery.

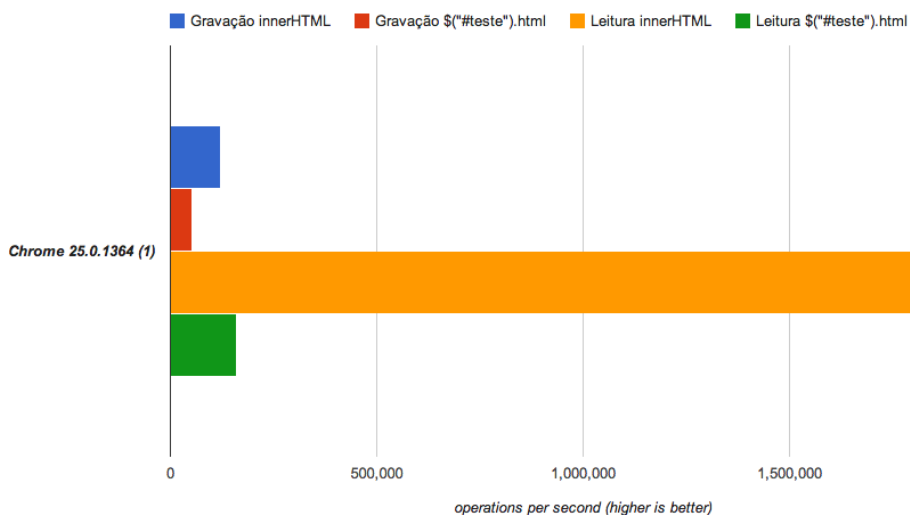



Figura 14.4: HTML - Operações por segundo (mais é melhor)

Na mesma linha, temos na figura 14.5 a diferença de performance entre duas instruções que retornam o `id` de um elemento.

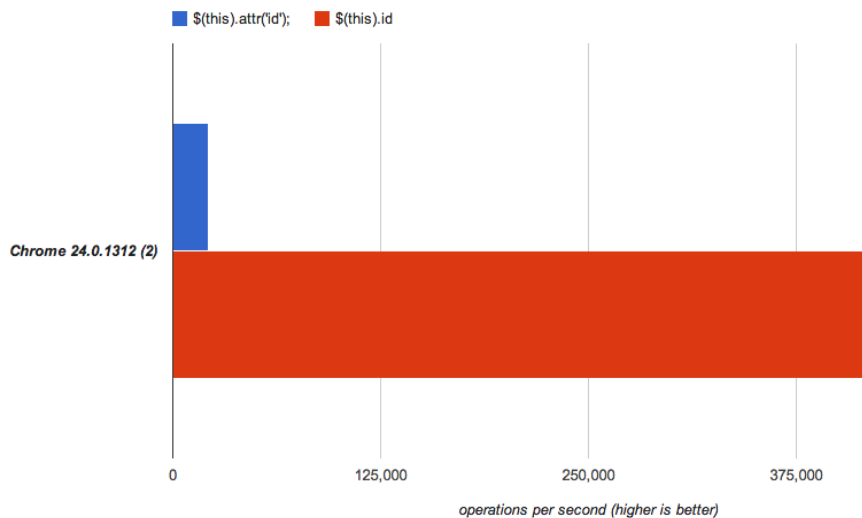



Figura 14.5: ID - Operações por segundo (mais é melhor)

Usamos o código abaixo para o teste:

```
var elemento = $("#teste");

$(elemento).attr("id");

elemento[0].id;
```

A primeira forma acessar o `id` após criar um objeto jQuery, enquanto a segunda acessa a propriedade diretamente no DOM.

Existem várias outras formas em que é preferível acessar diretamente o objeto DOM ou usar JavaScript puro. Na dúvida, crie seus próprios testes usando o jsPerf.

Capítulo 15

E o que vem agora?

15.1 Recomendações de leitura

No final do livro há uma lista de sites e livros que serviram de referência para sua composição.

Alguns livros foram muito úteis para tirar dúvidas e entender conceitos, mas não foram citados no decorrer dos capítulos. Por isso eu vou sugerir alguns deles para que você possa continuar estudando e avançando no aprendizado, tornando-se um profissional cada vez melhor e mais completo.

Infelizmente os livros recomendados aqui estão em Inglês, o que vai exigir um esforço a mais de sua parte. Eu não conheço nenhum título nacional que seja didático e que tenha conteúdo o bastante para que eu possa recomendar.

Mesmo entre os títulos em Inglês, por mais numerosos que sejam, fica difícil encontrar livros que não repitam sempre as mesmas coisas e fiquem naquele mais do mesmo.

JavaScript: The Good Parts

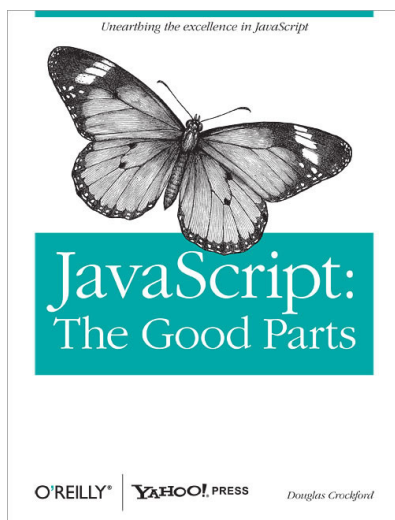



Figura 15.1: JavaScript: The Good Parts

Escrito por Douglas Crockford, o criador do JSON[4].

Eu considero esse o melhor livro já escrito sobre o assunto. No começo do livro eu comentei que o conteúdo poderia ser traumático para quem está começando. A boa notícia é que, agora que você passou por todos os capítulos, o livro do Crockford vai parecer bem simples de ler e entender.

Nele são apresentadas boas práticas e mesmo o jeito certo de se fazer as coisas em JavaScript, além de mostrar também técnicas de como trabalhar com JSON.

Secrets of the JavaScript Ninja

Escrito por John Resig, criador do jQuery, e Bear Bibeault[9].

Esse livro apresenta algumas técnicas de depuração e testes, expressões regulares e como lidar com threads. Se você quiser aprender mais sobre como utilizar técnicas avançadas com JavaScript dentro de um browser, recomendo a leitura.

Apenas como um comentário, a imagem da capa é de um samurai, e não de um ninja.

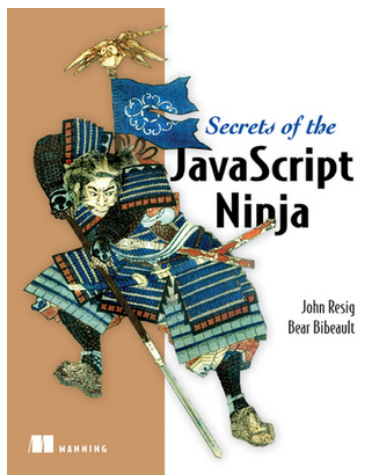



Figura 15.2: Secrets of the JavaScript Ninja

Pro JavaScript for Web Apps

Escrito por Adam Freeman, autor de outros dezenove livros pela mesma editora[6].

O mais interessante desse livro é que o autor demonstra técnicas que atualmente não são comuns no uso de JavaScript. Por exemplo, é demonstrado como utilizar MVC com JavaScript, como fazer seu site trabalhar offline e fazer a sua aplicação utilizar URLs personalizadas sem que isso seja enviado para o servidor e também como utilizar armazenamento local de dados em conjunto com HTML5.

Recomendo pelo conteúdo avançado e pouco usual, mesmo que você ache que dificilmente vá utilizar isso nas suas aplicações.

O livro é bem recheado com trechos de código e imagens, o que torna a fixação do conteúdo bem mais efetiva.

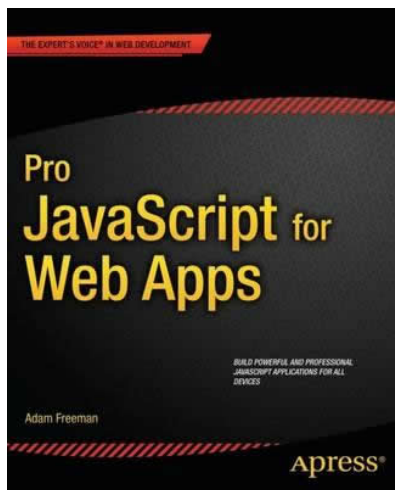



Figura 15.3: Pro JavaScript for Web Apps

15.2 Fim?

Sinta-se à vontade para buscar mais livros além dos que eu citei e para ir além do que foi mostrado neste livro.

O limite do conhecimento só depende do quanto de vontade e tempo você investir.

Caso queira entrar em contato comigo, ou com editora, acesse os links que eu disponibilizei no capítulo 1. Não pense duas vezes antes de enviar críticas, dúvidas, sugestões ou mesmo se quiser me pagar uma cerveja.

Boa sorte em seus projetos e obrigado pela confiança. Nos vemos por aí.

Índice Remissivo

Referências Bibliográficas

[1] Cisco. Cisco visual networking index. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html, 2013.

[2] W3 Consortium. Html: The markup language (an html language reference). http://www.w3.org/TR/html-markup/input.date.html, 2012.

[3] Douglas Crockford. Javascript: The world's most misunderstood programming language. http://www.crockford.com/javascript/javascript.html, 2001.

[4] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[5] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2007.

[6] Adam Freeman. *Pro JavaScript for Web Apps*. Apress, 2012.

[7] Jesse James Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications, 2005.

[8] Jesse James Garrett. About me. http://www.jjg.net/about/, 2006.

[9] Bear Bibeault John Resig. *Secrets of the JavaScript Ninja*. Manning, 2012.

[10] jQuery documentation. Authoring plugins. http://docs.jquery.com/Plugins/Authoring, 2013.

[11] "KorRedDevil". getelementsbyclassname implementation for internet explorer. http://bit.ly/RgTzlV, 2005.

[12] David Leavitt. *O homem que sabia demais*. Editora Novo Conceito, 2011.

[13] Sérgio Lopes. Não use jquery no seu site mobile: conheça o zepto.js. http://blog.caelum.com.br/nao-use-jquery-no-seu-site-mobile-conheca-o-zepto-js/, 2012.

[14] Mozilla. Gecko keypress event. https://developer.mozilla.org/en-US/docs/Gecko_Keypress_Event, 2008.

[15] Max Ogden. Callback hell. http://callbackhell.com/, 2012.

[16] Michael "sideshowbarker"Smith et al. Automatic semicolon insertion. http://es5.github.com/#x7.9, 2012.

[17] W3 Techs. Usage of javascript libraries for websites. http://w3techs.com/technologies/overview/javascript_library/all, 2010.

[18] Wikipedia. Ajax - history. http://en.wikipedia.org/wiki/Ajax_(programming)#History, 2012.

[19] Wikipedia. Same origin policy. http://en.wikipedia.org/wiki/Same_origin_policy, 2012.